Departamento de Ingeniería de Sistemas y Automática

PROYECTO FIN DE CARRERA
INGENIERÍA INDUSTRIAL

Desarrollo de Módulo de Visual Servoing para el Repositorio Open Source ASIBOT

Autor: Álvaro Martínez Estradé

Tutor: Alberto Jardón Huete
Director: Juan González Víctores

Leganés, Septiembre 2013

Título: Desarrollo de Módulo de Visual Servoing para el Repositorio Open Source ASIBOT
Autor: Álvaro Martínez Estradé
Tutor: Alberto Jardón Huete
Director: Juan González Víctores

EL TRIBUNAL

Presidente:

Vocal:

Secretario:

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día ..... de ..... de ... en ....., en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de:

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

Llegados a este punto tengo que estar agradecido a mucha gente que ha hecho que toda esta etapa de mi vida haya sido más llevadera y que fuera posible llevarla a buen término.

Primero y ante todo me gustaría mostrar mi mayor agradecimientos a mis padres que sin su apoyo constante hubiera sido imposible terminar la carrera, y junto con ellos a mis hermanos los cuales también me han dado fuerzas a lo largo de todos estos años.

Debo parte de mis logros también a muchos amigos, algunos solo con apoyarme y otros compartiendo interminables días de estudio en diferentes bibliotecas. En especial se merece mención Jorge Kazacos con cual he compartido toda la vida estudiantil y hemos sido un empujón el uno para el otro.

A la hora de realizar el proyecto tengo que dar las gracias a Alberto Jardón por ofrecerme realizar un proyecto dentro del departamento y sobretodo a Juan González Víctores que ha sido la persona que con la que más tiempo y esfuerzo he compartido a la hora de realizar este proyecto, mostrándose siempre dispuesto a ayudar.

Resumen

La finalidad de este proyecto es el desarrollo de un módulo software que permita el movimiento automático de un robot hacia un objeto dado, según las imágenes que proporcionará una cámara que llevará un robot en su efector final. En otras palabras, se busca desarrollar un módulo software que implemente un sistema de control Visual Servoing para robots. Este módulo se liberará como parte del repositorio software Open Source ASIBOT.

Al sistema se le indicarán cuatro puntos en coordenadas imagen, que formen un cuadrado, y que tendrá como referencia constante. Otro módulo se encargará de la segmentación de un objeto o varios del entorno que vea la cámara, de los cuales se obtendrán cuatro puntos que definan su cuadro delimitador. Con esto se inicia un bucle de control, que a cada paso irá encuadrando los puntos del objeto con respecto a los puntos de referencia a través del movimiento del robot. El bucle se detiene cuando el encuadre tenga un error menor a un umbral o cuando el bucle ha superado un número especificado de iteraciones.

Una vez finalizada la acción de control, se muestra el tiempo utilizado y se dibujan varias gráficas mostrando la evolución del error y de la posición de los motores a cada iteración del sistema. También se devuelven valores que indican la convergencia y la rapidez del sistema.

Palabras clave: Visual Servoing, Open Source, robótica, visión.

Abstract

The purpose of this proyect is the development of a software module, that allows the automatic movement of a robot towards a given object, based on images provided by a camera mounted on a robot end effector. In other words, it seeks to develop a software module that implements a Visual Servoing control system for robots. This module will be released as part of the Open Source software repository ASIBOT.

The system will be given 4 points, forming a square in image coordinates, having them as a constant reference. Another module will be responsible for the segmentation of one or various objects from the camera's visible environment, from which four points are obtained to define their bounding box. After this, a control loop is initiated, which at each step will be fitting the points of the object with regard to the reference points across the movement of the robot. The control loop stops when the setting error drops down below a certain threshold or when the loop reaches a previosly set number of iterations.

Once the control part is finished, we show the elapsed time and several graphs showing the evolution of both the error and the position of the engines associated to every iteration of the system. Also, the system returns values indicating the convergence and the quickness of the system.

Keywords: Visual Servoing, Open Source, robotics, vision.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo se explicará cual es la razón del proyecto, lo que abarca y su utilidad a la hora de resolver ciertos problemas. Definiremos los objetivos concretos, y avanzaremos la estructura del proyecto.

1.1. Motivación del proyecto

Vivimos en un planeta que cambia constantemente. Esto propicia que la humanidad, desde sus principios, se encuentre en constante evolución [1]. En un momento determinado, surgió el concepto de tecnología. Entendemos la tecnología como un conjunto de teorías y de técnicas que permiten el aprovechamiento práctico del conocimiento científico [2]. Esta nos permite satisfacer deseos y necesidades, es decir, nos facilitan la vida. Dentro de la tecnología, surgió la robótica, entendida como la técnica que aplica la informática al diseño y empleo de aparatos que, en sustitución de personas, realizan operaciones o trabajos, por lo general en instalaciones industriales [3]. La robótica aúna varias ramas de la tecnología de forma multidisciplinar.

En un principio, los robots se dedicaron al mundo industrial, donde hoy día están introducidos en mayor o menor medida según el sector. La ventaja de trabajar con robots es que éstos siempre realizan el trabajo de la misma manera y con la misma precisión,

sin cansarse, como ocurre sí con las personas. El mayor exponente podría ser la industria automovilística, donde realizan una gran variedad de tareas y con gran precisión. Los robots también se han introducido para la realización de tareas peligrosas, como puede ser la desactivación de bombas, exploración de volcanes o viajes espaciales. Los robots están cambiando la manera en la que se construye, se produce, se mantiene la seguridad, etc. Están expandiendo los límites de la humanidad en muchos sentidos. Podemos decir que vivimos en un mundo en constante evolución tecnológica, lo cual va mejorando la calidad de vida de las personas. Se estima que en España el 9 % de la población es discapacitada [4] y además esta tiene una relación directa con la edad. Dentro del marco de personas con discapacidad y/o ancianas, entendemos que necesitan una adaptación de las tecnologías para poder tener los mismos beneficios de esta que el resto de la población, además de poder tener las mismas oportunidades en la sociedad. Así surge la robótica asistencial, robots que ayudan a personas con cierta discapacidad a poder llevar una vida normal. Estos robots llevan una serie de sensores que les permite interactuar con un entorno cambiante.

En el mundo de la robótica tenemos diferentes sistemas de control para los robots. Debido a la aparición de multitud de tipos de sensores, se puede controlar un robot de muchas maneras diferentes. Las cámaras son de los sensores que producen mayor cantidad de datos sobre el entorno, como del propio robot. Un control de este tipo es el Visual Servoing, donde un robot se guía a través de una cámara. Este concepto es muy útil, ya que permite realizar el guiado de un robot hacia un objeto de forma autónoma. Introducir el control Visual Servoing en la robótica asistencial es muy interesante. Nos proporciona un intuitivo sistema de control, solo marcándole lo que deseamos buscar. De esta forma podemos conseguir que el robot nos dé de comer o de beber, sin tener que saber la posición en la que se encuentra el objeto. Sólo necesitará ver el objeto buscado

en la cámara y ya podrá guiarse hacia él, quedando a una distancia respecto a él que será fija y marcada por nosotros. Por otro lado, introducir un sistema de Visual Servoing en un robot asistencial o en cualquier otro, reduce su coste de implantación significativamente, ya que nos permite trabajar en entornos que no tengan que estar totalmente estructurados para el robot. El control Visual Servoing ofrece ventajas en comparación con el control visual en lazo abierto, como puede ser el hecho de la independiencia de la precisión de los sensores de visión, así como de la precisión del propio robot.

1.2. Objetivos

Este proyecto pretende implementar un sistema de control de velocidad en lazo cerrado mediante visión por ordenador para robots, es decir, se pretende aplicar un sistema de Visual Servoing. Primero se probará contra un simple robot cartesiano simulado, el cual será un "cubo". Se hará así porque todo el sistema calcula velocidades cartesianas y se podrán introducir directamente al robot. Una vez conseguido, se implementará el sistema en el entorno de la cocina de ASIBOT para conseguir guiar el robot hacia objetos de ésta.

Se quiere implementar, de la forma más parecida posible, el sistema de Visual Servoing basado en imagen desarrollado por Peter Corke en su libro [5] de robótica y visión *Robotics, Vision and Control: Fundamental algorithms in MATLAB*. El libro desarrolla un completo sistema de programas o toolbox en MATLAB, con lo que nuestro sistema de control Visual Servoing estará desarrollado también en MATLAB. Para poder aplicar todo este conocimiento primero tendremos que realizar un detallado estudio del libro mencionado, y así poder conocer el funcionamiento del sistema y sus requerimientos. Así podremos hacer las modificaciones necesarias cuando lo apliquemos a nuestro robot. Nuestros robots de prueba estarán simulados en OpenRAVE. Tendremos que ser capaces de conectar nuestro sistema de control (MATLAB) con el simulador

que se utlice (OpenRAVE), para lo cual se usará un servidor de comunicaciones (YARP). Intrínsecamente, el funcionamiento del sistema de control Visual Servoing requiere tener en paralelo sistemas de tratamiento de imágenes para poder detectar objetos del entorno. Tendremos que servirnos de trabajos previamente realizados en esta materia para el reconocimiento de objetos, y conectarlos con nuestro sistema de control para poder tener un robot completamente controlado.

1.3. Estructura del documento

A continuación, y para facilitar la lectura del documento, se detalla el contenido de cada capítulo.

- En el capítulo 2 se hace un repaso de todas las áreas que abarca el proyecto, es decir, el estado del arte. Debido a que la teoría es compleja, se hace un desarrollo de las ecuaciones en las que se basa el sistema de control. Se parte de las ecuaciones de las proyecciones en una cámara, hasta llegar a la ley de control que se busca. También se muestra la estructura de programas que se usa en la plataforma de simulación.

- El capítulo 3 se centra en la descripción del conjunto de programas que forma el sistema de control del robot. Se muestra el esquema organizativo y la jerarquía entre los programas realizados. Después se centra en cómo se utiliza el sistema de control por visión desarrollado.

- En el capítulo 4 se presentan una serie de experimentos junto con los resultados que proporciona. Además, en cada experimento se añaden gráficas para comparar el comportamiento del sistema según se varían ciertos parámetros. También se realizan experimentos de profundidad y de posición.

- En el capítulo 5 se muestran las conclusiones del proyecto,

así como posibles mejoras futuras que poder añadir al sistema para mejorar su funcionamiento.

Capítulo 2

Estado del arte

En esta sección del documento vamos a hablar primero del robot ASIBOT, para poder introducir su repositorio de forma más extensa. Después se tratarán los sistemas de control, en concreto, los aplicados a la robótica. Finalmente vamos a exponer todo lo relacionado con un sistema de control por Visual Servoing, donde profundizaremos en Image-Based Visual Servoing. Aparte del control por Visual Servoing, un sistema de simulación completo usará varias tecnologías distintas para poder funcionar, como programas de segmentación, servidores de comunicaciones, diferentes lenguajes de programación, y lo más importante, un robot. Vamos a introducir el concepto de guiado por visión, que es el principal de la investigación, en un robot asistencial, dando lugar a un robot asistencial mucho más autosuficiente y completo en sus funciones.

2.1. Robot ASIBOT

Entendemos que un brazo robótico es un brazo mecánico que está formado por una serie de eslabones, unidos mediante articulaciones, (las cuales pueden ser rotacional, prismática, cilíndrica, esférica o de tornillo) que realizarán una función parecida a la de un brazo humano. El último eslabón del brazo es conocido como efector final o end effector y es ahí donde se colocará el manipulador al brazo, que según la tarea que vaya a realizar el brazo puede ser desde

una mano o una ventosa, hasta la sujeción de una broca (figura 2.1).

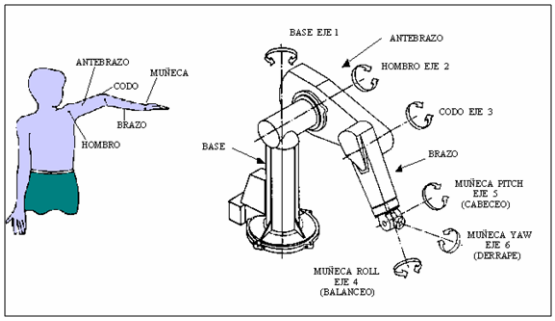

Figura 2.1: Robot tipo brazo

Dentro del mundo de los brazos robóticos, podemos distinguir los que están destinados al mundo industrial de los que están destinados al mundo personal. Los primeros están pensados para ser productivos y cuanto más eficientes mejor, es decir, buscamos que tengan un determinado rendimiento económico, por lo que suelen ser brazos grandes, con una infraestructura compleja. Suelen tener un entorno prefijado para el robot poder ser muy rápido a la hora de realizar sus tareas. En los robots personales no se busca de ellos un rendimiento económico. Por ejemplo los robots asistenciales, están pensados para ayudar a personas discapacitadas. Están diseñados para interactuar directamente con personas con cierta discapacidad, y poder producir cierta rehabilitación en ellas.

ASIBOT es un manipulador orientado a la robótica asistencial [10], es decir, es un robot orientado a la ayuda de personas y no al mundo industrial. El robot se puede ver en la figura 2.2. El robot tiene una serie de características especiales, como son su reducido peso (unos 10 kg), sistema de control a bordo y lógicamente una moderada capacidad de carga (2 kg). Tiene un alcance de 1.3 m y básicamente para ponerlo a funcionar basta con conectarlo a una fuente de 24V. ASIBOT tiene una configuración cinemática en cadena abierta simétrica, con 5 grados de libertad. El hecho de tener todo el sistema de control abordo, además de su reducido peso y

tamaño, le confiere a ASIBOT una portabilidad total.

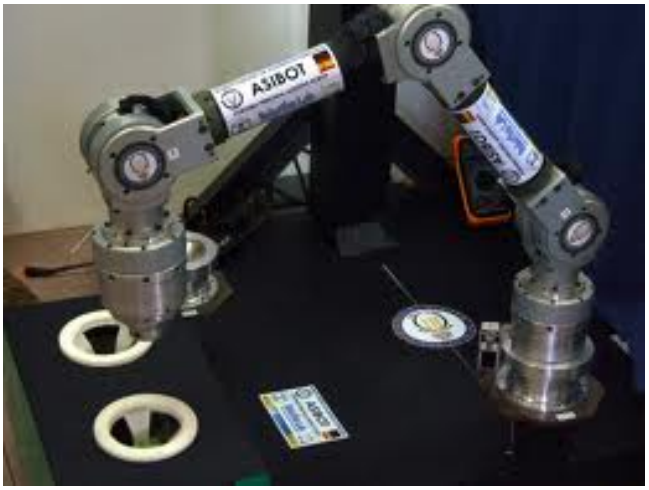

Figura 2.2: Robot ASIBOT

ASIBOT puede funcionar desde en una silla de ruedas, hasta ir moviéndose por estaciones de anclaje situadas en un determinado entorno. Al estar conectado en un extremo a una estación de anclaje, recibirá la energía de la propia estación, sin tener que realizar conexiones adicionales. Tiene una estructura simétrica y unos conectores cónicos en ambos extremos, que son los que le permitirán desplazarse por el entorno. Para interactuar con ASIBOT antiguamente se realizaba con una PDA con wireless. Hoy en día se puede interactuar con el robot mediante una interfaz web y comandos de voz [11] o más recientemente mediante gafas 3D. Podemos llegar a tener un funcionamiento en el que ASIBOT se mueva entre estaciones de anclaje hasta llegar a la requerida. Una vez en ella pasa a tener comportamiento de robot manipulador, conectándole el dispositivo necesario en el efector final.

2.2. Repositorio Open Source ASIBOT

Esta sección trata de la implementación del repositorio Open Source, así como de su manejo para la simulación de ASIBOT bajo cualquier sistema operativo Open Source Linux. El manual está realizado en Ubuntu pero no debería suponer un problema instalarlo en otra distribución o sistema operativo, ya que el repositorio es además multiplataforma. La dirección del repositorio es `http://roboticslab.sourceforge.net/asibot/`.

2.2.1. Simulación y control básico

Lo primero que hay que realizar es instalar las dependencias de CMake, YARP y OpenRAVE. Una pequeña introducción de OpenRAVE y YARP se muestra a continuación.

OpenRAVE

OpenRAVE (Open Robotics Automation Virtual Environment) [12] es un entorno libre para la simulación, modelado y planificación de robots. Es el simulador en el que probamos todo el sistema de guiado visual. En la figura 2.3 podemos ver uno de los entornos de trabajo modelado en los que vamos a trabajar. Se trata de una herramienta muy flexible, hasta el punto que uno le integra lo que quiere que contenga el simulador al cargar un determinado entorno (simulación de físicas, colisiones, menú propio...), y por supuesto se puede modelar el robot que se quiera probar.

Una de las tecnologías más impactantes que incluye OpenRAVE es IKFast, que es un compilador para cinemática. Al contrario que otros compiladores de cinemática inversa, puede resolver analíticamente las ecuaciones de la cinemática, independientemente de la complejidad de la cadena cinemática y genera archivos específicos para cada lenguaje (como C++) para su posterior uso. Los usuarios pueden concentrarse en la planificación y desarrollo de problemas sin tener que gestionar los detalles de cinemática y dinámica, colisiones,

actualizaciones del mundo, etc. El robot y el entorno, así como los objetos que contenga este, son modelados en XML. Al ser tan flexible, podemos ver por ejemplo una ventana con las imágenes que va viendo la cámara que lleva ASIBOT en el efector final. La arquitectura proporciona una interfaz que se puede usar en combinación con otros paquetes de robótica como Player y ROS, ya que se centra en la planificación del movimiento autónoma y secuencias de control a alto nivel más que en el bajo nivel.

Figura 2.3: OpenRAVE simulando el entorno de trabajo del robot ASIBOT

YARP

Una manera de comunicación entre programas es mediante sockets, los cuales genéricamente los podríamos definir como la manera en que dos programas pueden intercambiar información de una manera eficaz [13]. Los sockets al final son un conjunto de dos direcciones de red, una la del ordenador y otra la del puerto que abrimos en el programa. Para que la comunicación sea posible, necesitan usar el mismo protocolo ambos programas que queramos conectar. Para ampliar este concepto surge YARP.

YARP (Yet Another Robot Platform) es una herramienta o conjunto de ellas, junto con protocolos y librerías escrito en C++ que permite tener herramientas y módulos desacoplados. Es un software

libre y gratuito desarrollado por RobotCub Consortium. En general el mundo de la robótica está en constante evolución, tanto hardware como software, mediante YARP podemos conseguir que los proyectos de robótica puedan ser más duraderos en el tiempo, ya que nos permite tener partes del proyecto independientes entre sí, pudiendo evolucionar las partes sin afectar al conjunto. YARP incluye un modelo de comunicación que es neutro para transporte, de manera que el flujo de datos está desacoplado de los detalles de las redes y protocolos en uso subyacentes.

Está diseñado para poder operar con otras arquitecturas. Se pretende facilitar el intercambio de código entre investigadores, sobre todo cuando esto reduce el tiempo necesario para crear una plataforma y utilizarla para investigar. La comunicación entre los distintos módulos se realiza mediante un servidor virtual que crea YARP, al que nos podemos conectar mediante puertos que abriremos en YARP y en los distintos módulos que requieran cruzar datos (figura 2.4). En la red, la interoperabilidad básica se puede realizar con unas pocas líneas de código en cualquier lenguaje de programación con una librería de puertos (sockets), la interoperabilidad máxima se conseguirá siguiendo los protocolos documentados. Para que todo esto sea posible YARP se compila mediante CMake, el cual hace los proyectos fáciles de compilar dentro de una amplia gama de entorno de desarrollo integrado, sistemas Unix, código Apple, Microsoft Visual C++, etc. Para las funciones dependientes del sistema operativo usa las librerías de código abierto ACE. SWIG toma el código fuente de C/C++ y genera envoltorios para él, utilizables en distintos lenguajes de programación. La documentación se ha extraído de un documento de YARP [14].

Instalación del repositorio Open Source ASIBOT

Una vez tenemos instaladas dichas dependencias vamos a instalar la simulación y el software de control básico. Para ello vamos a utilizar la terminal de Ubuntu. Vamos a teclear lo siguiente por

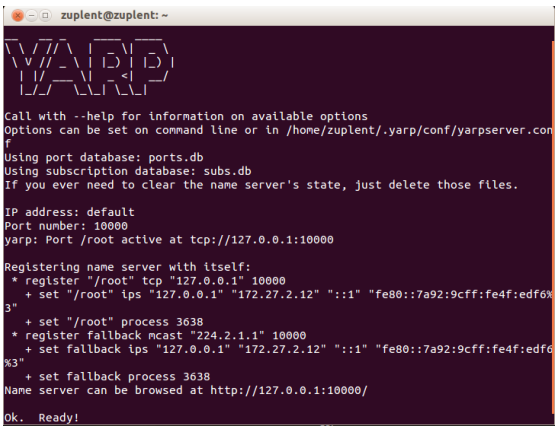

Figura 2.4: Servidor YARP en terminal de Linux

terminal.

```
svn co https://svn.code.sf.net/p/roboticslab/code/asibot
    asibot;
cd asibot/main; mkdir build; cd build;
cmake .. -DENABLE_cartesianServer=ON -
    DENABLE_RlPlugins_cartesianbot=ON -
    DENABLE_RlPlugins_ravebot=ON;
make -j3; sudo make install; cd ../../..
```

Ahora hay que apuntar la variable de entorno ASIBOT_ROOT a "asibot" y la variable de entorno ASIBOT_DIR a "asibot/main/build". Para realizar esto tenemos dos opciones:

- A través de ∼/.profile (analizado al reiniciar).

```
echo "export ASIBOT_ROOT=$PWD/asibot" >> ~/.profile;
echo "export ASIBOT_DIR=$PWD/asibot/main/build" >> ~/.
    profile;
source ~/.profile;
```

- A través de ∼/.bashrc (analizado dentro de cada terminal).

```
echo "export ASIBOT_ROOT=$PWD/asibot" >> ~/.bashrc;
echo "export ASIBOT_DIR=$PWD/asibot/main/build" >> ~/.
    bashrc;
source ~/.bashrc;
```

Una vez hecho esto podemos pasar a instalar la carpeta de las aplicaciones.

```
cd $ASIBOT_DIR
make install_applications
```

Y después instalar cada aplicación específica, como por ejemplo.

```
cd $ASIBOT_DIR/app/visualServo/scripts
cp visualServo3.xml.template visualServo3.xml
```

Ahora ya tenemos instalado el repositorio básico de ASIBOT, podemos mostrar la simulación y control básicos del mismo.

Manejo básico

La implementación actual utiliza YARP para comunicaciones. El manejo básico de YARP requiere el uso de un servidor centralizado. Este servidor asocia la implementación de bajo nivel de los puertos de comunicaciones con los nombres que les ponemos. Antes de ejecutar cualquier módulo de ASIBOT hay que lanzar un servidor de YARP.

```
[terminal 1] yarp server
```

El simulador que usamos para ASIBOT es versátil y multicapa. Por comodidad, se ha montado un pack básico del simulador robótico en el espacio articular con un solver de cinemática inversa y un controlador para la misma que viene con los parámetros por defecto. Se puede lanzar con un solo comando:

```
[terminal 2] cartesianServer
```

Ahora deberá saltar una ventana parecida a la siguiente (figura 2.5).

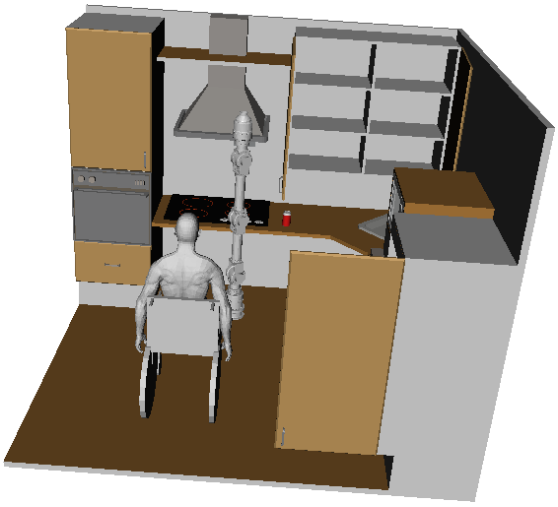

Figura 2.5: Instancia del módulo cartesianServer

El simulador se puede lanzar con distintos parámetros y así cambiar los que vienen por defecto. Cada capa de los módulos de ASIBOT tiene parámetros que se pueden cambiar de tres maneras distintas. Los parámetros se introducen en el siguiente orden (la segunda forma sobrescribe la primera, y la tercera sobrescribe ambas).

- Parámetros por defecto definidos en los archivos de cabecera de la clase (archivos con extensión *.h o *.hpp). Se debe recompilar el proyecto si se cambia cualquiera de estos parámetros.
- Archivos de configuración (archivos con extensión *.ini o *.xml). Estos archivos son bajados de $ASIBOT_ROOT/main/app y son instalados en $ASIBOT/app con el comando ya citado make install_applications. Dentro de esta carpeta, hay carpetas separadas para los módulos y las capas de los módulos.
- Línea de comandos (en la ejecución del programa). Los parámetros son modificados al ejecutar un programa, utiliza el

siguiente formato: ./program - -parameter new_value

Por ejemplo, suponiendo que nuestra tarjeta gráfica nos lo permite hacer y queremos cargar un entorno que contenga cámaras. Podemos ejecutar el programa con el siguiente comando de manera que cambiamos los parámetros en la misma línea.

```
[terminal 2] cartesianServer --env asibot_kitchen_cameras.
    env.xml
```

Este entorno contiene cámaras simuladas (figura 2.6). Más adelante se verá como realizar las conexiones.

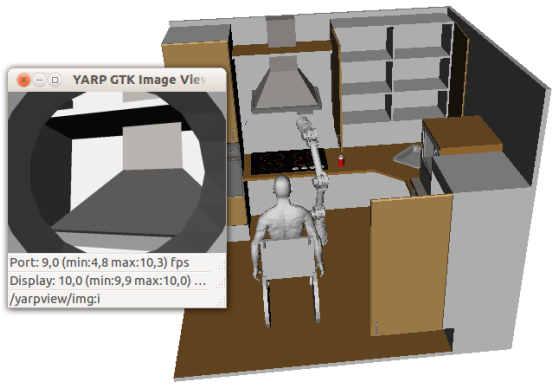

Figura 2.6: Instancia del módulo cartesianServer con cámaras

Además podemos ver los parámetros por defecto y los seleccionados, con el parámetro - -help.

```
[terminal 2] cartesianServer --env asibot_kitchen_cameras.
    env.xml --help
```

Si queremos afectar a un parámetro de una forma más persistente, podemos cambiar el archivo de configuración. Para este caso en concreto, el archivo de configuración de cartesianServer se encuentra en $ASIBOT_ROOT/app/cartesianServer/conf/cartesianServer.ini. En este archivo podemos ver que prácticamente todos los parámetros se encuentran comentados (con los caracteres //). Esto es una

convención para indicar que esos son los parámetros por defecto indicados en las cabeceras. Para realizar la misma operación debemos buscar y cambiar esta línea.

```
// env asibot_kitchen.env.xml        /// env [xml]
    environment name in abs or rel
```

Por esta otra sin comentar.

```
env asibot_kitchen_cameras.env.xml   /// env [xml]
    environment name in abs or rel
```

2.2.2. Control Cartesiano

En esta sección vamos a ver cómo interactuar con el módulo de control cartesiano y que nos permite hacer. Con este control, el robot se puede mover linealmente en la dirección de X, Y, Z.

CartesianServer

Primero debemos lanzar una instancia de YARP y otra de cartesianServer igual que como se ha mostrado antes.

```
[terminal 1] yarp server
```

```
[terminal 2] cartesianServer
```

El módulo cartesianServer actúa como la parte servidor de una envoltura de red de la clase CartesianBot mediante la clase CartesianServer. La implementación mapea ciertas llamadas a función de YARP rpc's a CartesianBot. Por lo tanto se puede interactuar con la clase desde la línea de comandos. Una vez tenemos funcionando el servidor y el módulo, abrimos una nueva terminal para interactuar con el módulo cartesianServer y tecleamos lo siguiente:

```
[terminal 3] yarp rpc /ravebot/cartesianServer/rpc:i
```

Sólo con cambiar 'ravebot' por 'canbot' controlamos al robot real. El uso de corchetes [ ] indica que estamos enviando un VOCAB, que es un tipo de variable especial de YARP que nos permite enviar hasta 4 caracteres ASCII dentro de un char. El uso de paréntesis () indica que estamos enviando una lista, que es un Bottle dentro de un Bottle. Un Bottle es una clase especial de YARP para enviar y recibir datos entre los distintos módulos. Puede contener VOCABs y listas en su interior, por ejemplo una lista de variables double. Se envían posiciones u orientaciones como una lista de 5 elementos, correspondientes a los 5 grados de libertad de ASIBOT, x[m], y[m], z[m], rot(y')[grados], rot(z")[grados] del efector final en coordenadas absolutas de la base. A continuación se muestran los comandos RPC que se pueden ejecutar desde esta conexión (terminal 3).

```
[inv] (0.3 0.3 0.7 90 0)
```

Obtendremos como respuesta algo similar a:

```
(45.0 -41.169914 116.855705 14.314209 0.0) [ok]
```

Que es una respuesta de la cinemática inversa, pero sin producir movimiento, es decir, los valores por articulación que son necesarios para llegar a la posición indicada. La tabla 2.1 muestra el resumen de los diferentes tipos de comandos que podemos mandar:

Tabla 2.1: Comandos RPC

comandos rpc	respuesta ejemplo
[movj] (.1 .1 .7 90 0)	[ok]
[movl] (.1 .3 .8 90 0)	[ok]
[stat]	(0.0 0.0 1.4 0.0 0.0) [ok]
[stop]	[ok]

El comando [movj] produce movimiento con interpolación en el espacio articular del robot. Mientras que el comando [movl] produce

movimiento con interpolación pero en el espacio cartesiano. El comando [stat] devuelve la posición actual cartesiana del efector final. Por último el comando [stop] produce la parada del robot.

La aplicación también tiene asignados ciertos comandos streaming YARP a las llamadas de la clase CartesianBot. Por lo tanto, podemos interactuar con la clase desde la línea de comandos tecleando en una nueva terminal:

```
[terminal 4] yarp write ... /ravebot/cartesianServer/
    command:i
```

La tabla 2.2 muestra los comandos disponibles:

Tabla 2.2: Comandos streaming

streaming command format
[bkwd] (0.0 90.0)
[fwd] (0.0 90.0)
[rot] (0.0 90.0)
[vmos] (0.0 1.0 0.0 0.0 0.0)
[pose] (0.0 1.0 0.0 0.0 0.0)

El comando [bkwd] hace un seguimiento de un punto virtual detrás del efector final (punto(rot(z))[grados/s] rot(y')[grados]). El comando [fwd] hace un seguimiento de un punto virtual delante del efector final (punto(rot(z))[grados/s] rot(y')[grados]). El comando [rot] realiza un seguimiento de la orientación del punto virtual (punto(rot(z))[grados/s] rot(y')[grados]). El comando [vmos] envía comandos de velocidades directos en el espacio cartesiano. El comando [pose] envía posiciones directas en el espacio cartesiano.

Otros comandos RPC

Primero necesitamos al igual que en el caso anterior lanzar una instancia de YARP y otra de cartesianServer.

```
[terminal 1] yarp server
```

```
[terminal 2] cartesianServer
```

Ahora vamos a interactuar a través de la línea de comandos mediante otra conexión, vamos a teclear lo siguiente.

```
[terminal 3] yarp rpc /ravebot/rpc:i
```

Podemos enviar posiciones absolutas en el espacio de las articulaciones, como por ejemplo.

```
[terminal 3] set poss (115.0 -45.305776 10.552447 29.803438
    30.0)
```

Se obtendrá una respuesta del tipo.

```
[ok]
```

La clase RaveBot también puede actuar con WorldRpcResponder que nos proporciona una forma de interactuar con el entorno. Podemos interactuar de la siguiente manera.

```
[terminal 4] yarp rpc /ravebot/world
```

Es posible crear cuadrados o esferas estáticas (no afectadas por la gravedad) mediante los siguiente comandos:

```
[terminal 4] world mk sbox .1 .1 .1 2 1.5 1
```

```
[terminal 4] world mk ssph .05 2 1.5 1
```

Mediante estos comandos se crea un cubo de $1\,m^3$ y una esfera de radio $0.05\,m$, ambos situados en el punto $(2, 1.5, 1)\,m$ del espacio. Se pueden agarrar estos objetos creados, por ejemplo agarramos la esfera creada mediante.

```
[terminal 4] world grab ssph 1 1
```

Para soltar el objeto agarrado se hace tecleando lo siguiente.

```
[terminal 4] world grab ssph 1 0
```

También se pueden agarrar objetos preexistentes, por ejemplo en el entorno de la cocina de ASIBOT podemos agarrar la lata que se encuentra en la encimera.

```
[terminal 4] world grab obj redCan 1
```

Que se suelta con el siguiente comando.

```
[terminal 4] world grab obj redCan 0
```

La clase RaveBot puede retransmitir datos de cada cámara o sensor de profundidad que encuentra. Si se ha llamado al entorno de cartesianServer con las cámaras como se ha explicado antes, se puede conectar con la cámara IP de la encimera mediante el siguiente comando.

```
[terminal 5] yarpview /yarpview/img:i &
[terminal 5] yarp connect /ravebot/ip_camera/img:o /
    yarpview/img:i
```

2.2.3. ColorSegmentor

Dentro del repositorio de ASIBOT también tenemos la aplicación colorSegmentor. Se trata de una aplicación que lanzada junto con un módulo cartesianServer realiza en el entorno simulado de ASIBOT con cámaras una segmentación básica y etiquetado a través de la cámara usada. El módulo funciona con las librerías de OpenCV. OpenCV es una biblioteca compuesta por una serie de librerías de software libre para la visión por ordenador. Contiene una gran variedad de funciones en el ámbito del procesamiento de imágenes,

podemos encontrar desde sistemas de seguridad con detección de movimiento hasta aplicaciones de control que requieran reconocimiento de objetos. Incluye una completa documentación en su web y en sus libros [15]. En sus inicios fue desarrollado por Intel pero hoy en día está bajo licencia BSD y lo mantiene Itseez.

Es una biblioteca multiplataforma con versiones para Linux, Mac OS X, Windows, iOS y Android. Está escrito en C y C++, lo cuál le proporciona una gran eficiencia, y además aprovecha las características multinúcleo de los ordenadores de hoy día, haciéndolo aún más eficiente y remarcando su enfoque hacia tareas en tiempo real. Contiene interfaces de C, C++, Python y Java. Dentro del módulo colorSegmentor también se puede realizar un etiquetado de regiones y marcar las etiquetas máximas que buscamos, devolviéndonos los centroides de las etiquetas realizadas. La aplicación es instalada en el conjunto de la instalación de ASIBOT. Ahora necesitamos adaptar la plantilla (template) de $ASIBOT _ROOT/app/colorSegmentor/scripts/ y adaptar el archivo colorSegmentorSim.xml.template a nuestras necesidades mediante.

```
cd $ASIBOT_ROOT/app/colorSegmentor/scripts
cp colorSegmentorSim.xml.template colorSegmentorSim.xml
```

Para trabajar con este módulo tenemos que realizar lo siguiente.

```
[terminal 1] yarp server
```

Y ahora en otra terminal debemos ir a $ASIBOT_ROOT/app y lanzar (g)yarpmanager, donde la aplicación colorSegmentor debe estar disponible.

```
[terminal 2] cd $ASIBOT/app
[terminal 2] gyarpmanager
```

2.3. Sistemas de control en robótica

Un sistema de control es el formado por unas determinadas componentes que son capaces de regular su comportamiento con el fin de lograr un funcionamiento determinado mediante las entradas, es decir, el objetivo es controlar las salidas del sistema de alguna forma prescrita mediante las entradas a través de los elementos de control. Existen dos tipos de sistemas de control: en lazo abierto y en lazo cerrado.

Sistema de control en lazo abierto. Es el tipo de sistema de control en el que la salida del mismo no afecta al sistema de control. En este tipo de sistemas no se realimenta la salida, por lo que no la medimos. La salida no se compara con la entrada de referencia por lo que ante la misma entrada de referencia, la salida siempre será la misma. La precisión depende de la calibración previa del sistema. Un sistema de este tipo seguirá el esquema de la figura 2.7.

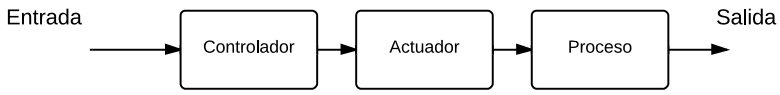

Figura 2.7: Sistema de control en lazo abierto

Sistema de control en lazo cerrado. Es el tipo de sistema de control en el que la salida del mismo afecta al sistema de control del sistema, es decir, tenemos una realimentación. Al tener una realimentación necesitamos medir la salida, que una vez medida comparamos con la entrada de referencia del sistema. Se usa la dife-

rencia entre ambas como medio de control. Una misma entrada no producirá siempre la misma salida en el sistema. Un sistema de este tipo seguirá el esquema de la figura 2.8.

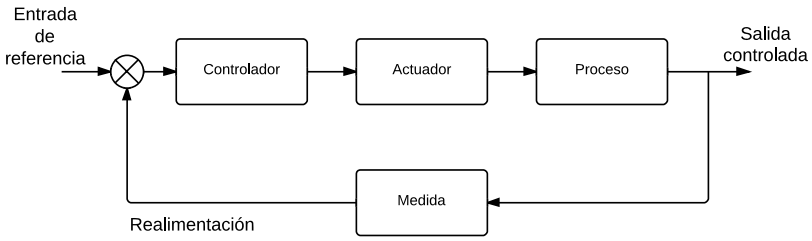

Figura 2.8: Sistema de control en lazo cerrado

2.3.1. Control en robótica

En el libro *Robotics: modelling, planning and control* [6], se explica que un sistema de control para un sistema robótico debe estar dotado de las siguientes características:

- Capacidad de mover objetos en el entorno de trabajo, es decir, habilidad de manipulación.
- Capacidad de obtener información del estado del sistema y del entorno de trabajo, es decir, que tenga habilidad sensorial.
- Capacidad de explotar la información para modificar el comportamiento del sistema de manera programada, habilidad de la inteligencia.
- Capacidad de guardar, elaborar y proveer datos de la actividad del sistema, que tenga habilidad de procesar datos.

Una forma efectiva de conseguir implementar todas las funciones, se puede realizar con una arquitectura funcional, que está pensada

como la superposición de diferentes niveles de actividad organizados en una estructura jerárquica. Los niveles más bajos de la arquitectura están dedicados al control del movimiento físico del robot, mientras que los niveles más altos se dedican a la planificación lógica de la acción. Los niveles están conectados por flujos de datos, en donde los niveles más altos manejan datos de medidas de sensores y/o resultados de acciones, mientras que los niveles más bajos manejan datos de transmisiones o direcciones. El modelo de referencia podría ser como en la figura 2.9.

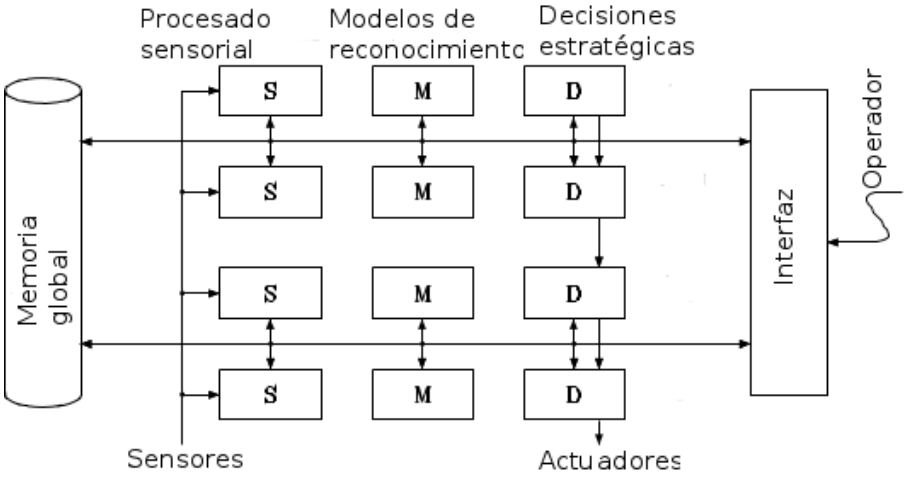

Figura 2.9: Sistema de control con arquitectura funcional

Normalmente, sin tener en cuenta el tipo de manipulador mecánico que sea el robot, la especificación de la tarea (movimiento y fuerza del efector final) es llevada a cabo en el espacio cartesiano con respecto al efector final, mientras que las acciones de control (fuerzas generalizadas en el actuador de articulaciones) son llevadas a cabo en el espacio articular. En ambos casos esto nos lleva a estructuras de control en lazo cerrado para explotar los beneficios de la realimentación. Debido a la existencia de muchos tipos de sensores, existen muchos tipos de sistemas de control para robots. De los sistemas de control a alto nivel podemos destacar:

- Sistemas de control por **posición**. En estos sistemas se controlan los movimientos compatibles del robot con una realimentación mediante sensores de posición.

- Sistemas de control por **velocidad**. Este tipo de sistemas controlan los movimientos compatibles del robot con una realimentación proporcionada por sensores de velocidad, es decir, tenemos en cuenta el avance/retroceso del robot en alguna dirección.

- Sistemas de control híbrido **fuerza-posición** [7]. En estos sistemas presentan un enfoque conceptual híbrido para controlar movimientos compatibles de un robot manipulador. La técnica híbrida combina información de fuerza y par con los datos de la posición, para satisfacer simultáneamente limitaciones en la trayectoria de fuerza y posición definidas para una tarea dada y en un sistema de coordenadas dado. Simulación, análisis y experimentos se usan para evaluar la capacidad del controlador para ejecutar dichas trayectorias usando realimentación de sensores de fuerza y posición que se encuentran en las articulaciones.

- Sistemas de control híbrido **fuerza-velocidad**. Este tipo de control se realiza la realimentación con datos de fuerza y de velocidad. Por ejemplo, se obtienen las restricciones de la realimentación lo que permite especificar tareas compatibles en función de una fuerza deseada (normal o tangencial) y de una velocidad deseada (normal o tangencial). Esto sirve para tareas que requieran por ejemplo movimiento ortogonal a la fuerza de contacto, como puede ser girar una manivela o rastrear una superficie.

- Sistemas de control por **impedancia** [8]. La manipulación, según el caso, tiene unas limitaciones determinadas que nos

dice que tener en cuenta la posición o la fuerza como vector de control es insuficiente. Se necesita un control de tal manera que el manipulador presente el comportamiento de una impedancia. Básicamente se busca que la relación entre los movimientos y el comportamiento dinámico comandados se puedan representar por una red equivalente Norton.

- Sistemas de control **visual feedback**. Tener robots guiados por visión ha sido y es cada día más una de las mayores áreas de investigación dentro del mundo de la robótica. Tradicionalmente la detección y manipulación visual están integrados en forma de lazo abierto, "mirar" y entonces "mover". La precisión de la operación depende entonces de la precisión del sensor visual, del manipulador y de su controlador. Estos sistemas se denominaban sistemas "visual feedback".

- Sistemas de control **Visual Servoing**. Una alternativa para mejorar la precisión de los sistemas visual feedback consiste en usar un control con retroalimentación visual, lo que aumentará la precisión general del sistema. Visual Servoing se refiere a tener robots guiados mediante una realimentación extraída de una cámara, es decir, realimentación por visión [9]. El término Visual Servoing resulta en la unión de muchas áreas elementales, como el procesamiento de imágenes a alta velocidad, cinemática, dinámica, teoría de control y computación en tiempo real. Algunos sistemas robóticos que incorporan visión están diseñados para la programación a nivel de tarea [21]. Son sistemas generalmente jerárquicos, en los que el nivel más alto es capaz de resolver la tarea con un modelo del entorno dado. Lo primero que hacen es calibrar la localización del objetivo y del lugar de agarre, mediante visión estéreo o telémetros láser. Entonces se planifica la secuencia de movimientos necesaria para completar la tarea.

2.4. Visual Servoing

El concepto de Visual Servoing es relativamente nuevo, los pioneros en conseguir hacer funcionar un concepto similar fueron Shirai y Inoue [16]. Más tarde uno de los primeros documentos sobre la materia fue escrito por Gerald J Agin [17], en donde ya se habla de Visual Servoing. El primer tutorial sobre el tema surgió a mediados de los 90 [18]. El control y seguimiento visual están cada día más en auge gracias al aumento de potencia de los algoritmos y sobre todo de los ordenadores. En parte, gracias a eso, surgieron tutoriales más modernos como el que escribió F.Chaumette [19].

El control por Visual Servoing se refiere a la utilización de datos de visión por computador para controlar el movimiento de un robot. Como se menciona en el escrito de Peter Corke [20], el control visual de manipuladores promete substanciales ventajas cuando se trabaja con objetivos cuya posición es desconocida o variable en el tiempo, manipuladores imprecisos o que requieran cierta flexibilidad.

Visual Servoing no es más que el uso de visión al más bajo nivel, con procesamiento sencillo de imágenes, para dar un comportamiento reactivo o reflexivo. Tiene mucho en común con la visión por computador activa [22]. Esta propone que una serie de comportamientos visuales pueden realizar la tarea a través de la acción, como puede ser el control de la atención. El fundamento principal de la visión activa no es interpretar la escena y luego modelarla, sino centrar la atención en la parte de la escena que es relevante para la tarea en cuestión. Si el sistema quiere aprender algo, en vez de consultar un modelo, consulta directamente al mundo, apuntando hacia él sus sensores.

La bibliografía relacionada con la estructura del movimiento también es importante para el Visual Servoing. En robótica generalmente tenemos información a priori del objeto de estudio y la relación

espacial entre las características con las que marcaremos dicho objeto (también llamadas features) es conocida. Por ejemplo, si se trabaja con un cuadrado, la información a priori es que la distancia que tiene que haber entre los vértices se mantiene.

Problema principal

El problema principal al que se enfrenta un sistema de Visual Servoing consiste en controlar la posición del efector final del robot, con respecto a un objeto concreto de su lugar de trabajo, utilizando para ello características visuales extraídas de la imagen. La cámara tiene una lente que realiza una proyección 2D de la escena en el plano de la imagen. Esta proyección hace que se pierda información. Cada punto en el plano de la imagen corresponde a un rayo en el espacio 3D. Se necesita cierta información adicional para poder determinar la coordenada 3D de un punto en el plano de la imagen. Esta información vendrá dada por múltiples puntos de vista, o por el conocimiento de la relación geométrica entre varios puntos feature del objetivo.

Una vez detectado en la imagen el objeto buscado, lo marcaremos en esta de una determinada manera, que serán uno o varios puntos, y serán nuestros puntos de interés o features. Una feature es definida generalmente como cualquier relación medible en una imagen. Las más usadas comúnmente son las coordenadas de puntos o centroides de regiones. Todo el sistema funcionará alrededor de estas features marcadas en la imagen. Tres o más features pueden ser usadas para determinar la posición de un objeto respecto a la cámara, con un conocimiento de la relación geométrica entre ellos. Se le marca al sistema una posición a la que se quiere llegar, en la cual las features tendrán unas determinadas coordenadas en la imagen que ve la cámara, y se van a comparar con las que vea la cámara en su posición inicial y a cada iteración, para conseguir un movimiento hacia el objeto, que será de acercamiento o de alejamiento hasta llegar a

la posición marcada.

Dentro de este primer acercamiento surgen ya dos posibilidades: tener la cámara montada en el efector final del robot, o tener la cámara fija en algún punto del entorno de trabajo del robot. Si la cámara va montada en el robot, tendrá que ir montada en su efector final, si el robot por ejemplo es un coche cualquier punto de la base del coche es el propio efector final. Sin embargo, si es un brazo robótico, el efector final será el último eslabón del mismo ya que con éste es con el que realizará el trabajo que sea y donde tendrá o no montadas sus herramientas.

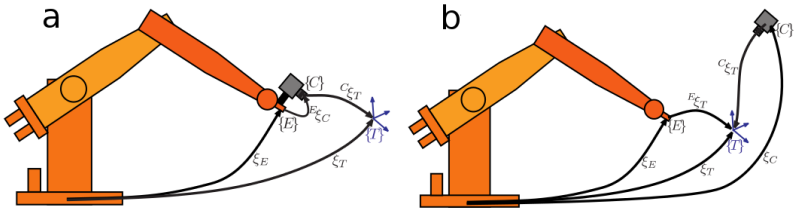

Figura 2.10: Configuraciones de visual servoing

La configuración de la figura 2.10.a tiene la cámara fijada en el efector final del robot desde donde observa el objetivo, y es conocido como **eye-in-hand o punto final en lazo cerrado**. La configuración de la figura 2.10.b tiene la cámara fijada en un punto del mundo, desde donde observa el objetivo y al propio robot, esta configuración es conocida como **punto final en lazo abierto**.

Otro criterio para la clasificación del control visual, distingue entre si utilizamos control dinámico de bajo nivel del propio robot o no. Decimos que tenemos control **indirecto** si el lazo de control envía las velocidades en referencias cartesianas al control de articulación del robot. Tenemos control **directo** si en el lazo de control tenemos desarrollado un controlador específico que actúa directamente sobre los motores del robot, enviando señales de tensión o par desde el control. El control indirecto simplifica la ley de control,

mientras que con el control directo se puede conseguir una respuesta más rápida del sistema, pero es más complejo a la hora de conseguir que el sistema sea estable.

Enfoques eye-in-hand

En este tipo de sistemas el robot lleva la cámara encima, y nos permite realizar otra clasificación según el control. Fue introducido por Sanderson y por Weiss [23]. En estos tipos de control al llevar la cámara en el efector final del robot, el movimiento del robot producirá movimiento en la imagen, lo que hace que sea totalmente distinto de los enfoques con cámara fija en el mundo. Hay dos enfoques fundamentalmente distintos para el control por Visual Servoing. Por un lado vamos a tener Visual Servoing basado en la posición, "Position-Based Visual Servo" (PBVS), y por el otro lado tenemos Visual Servoing basado en la imagen, "Image-Based Visual Servo" (IBVS). En general, si tenemos por ejemplo cuatro features marcadas en la imagen que forman un cuadrado, queremos conseguir una acercamiento dentro del plano de la imagen como el que se ve en la figura 2.11. En la imagen los puntos marcados con círculos

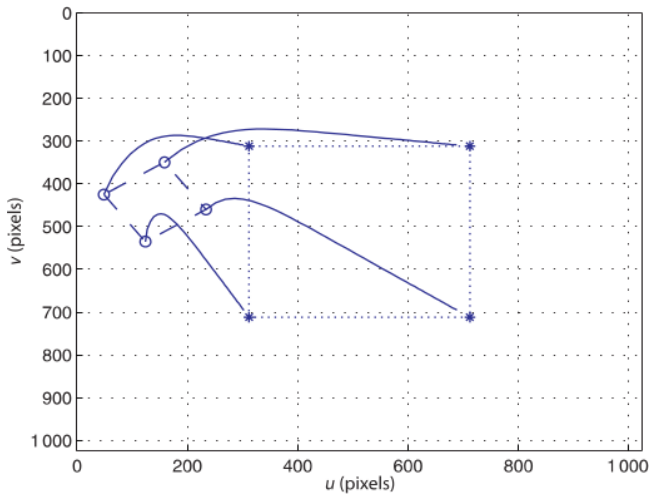

Figura 2.11: Funcionamiento Visual Servoing visto por la imagen de la cámara

son la vista inicial de la cámara y los puntos marcados con asterisco (*) son la vista final que queremos que vea la cámara, de manera

que para llegar a ellos los puntos tendrán que seguir una trayectoria similar a la marcada en la imagen.

PBVS

El Visual Servoing basado en posición usa las características sacadas de la imagen o features, además de una cámara calibrada y un modelo geométrico conocido del objetivo para poder determinar la posición 3D del objetivo con respecto a la cámara. Para poder realizar esto va a necesitar la información de la profundidad de los puntos. Los seres humanos utilizamos una amplia variedad de señales para conseguir esa información, como pueden ser las texturas, sombras, paralelismos, etc. Aproximaciones válidas para visión por computador fueron desarrolladas por Jarvis [24].

La información se podrá obtener por sensores de rango activo que proporcionan información de profundidad y orientación. Por otro lado se pueden usar técnicas pasivas como pueden ser técnicas fotogramétricas, visión estéreo con dos cámaras que interpretan la misma escena, técnicas de extracción de la profundidad a través del movimiento o técnicas de extracción de profundidad a través de la dinámica. Una vez obtenida la información necesaria el robot se mueve hasta la posición determinada, mientras el control se realiza normalmente en el espacio especial euclídeo de dimensión 3 o mejor denotado como SE(3). Se realiza la realimentación con la posición de las features en el espacio 3D. Existen buenos algoritmos para la estimación de la posición, pero son caros computacionalmente hablando y dependen en gran medida de la exactitud con la que se calibre la cámara y el modelo geométrico. Este control está ilustrado en la figura 2.12.a.

IBVS

Fue propuesto en un principio por Weiss and Sanderson [25] en 1983. En este tipo de Visual Servoing basado en imagen se omite

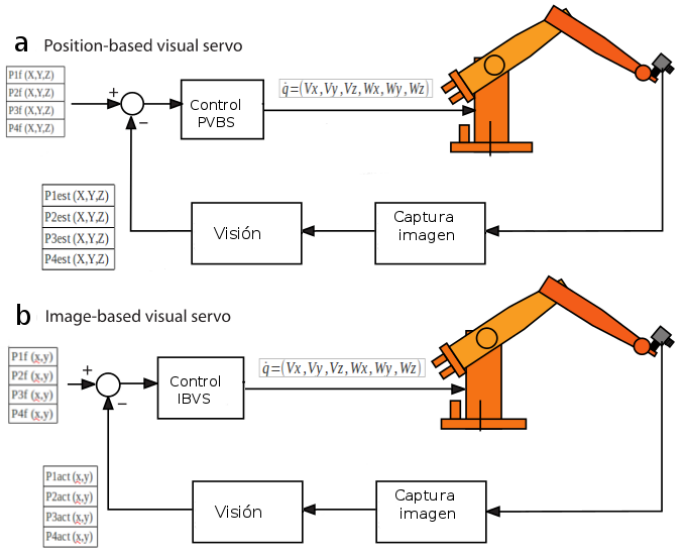

Figura 2.12: Tipos de sistemas de control de Visual Servoing

la estimación de la posición 3D y se usan características 2D de la imagen directamente. IBVS usa directamente la localización de las features en la imagen para la realimentación, es decir, los elementos de la tarea se especifican directamente en el espacio de la imagen y no en el mundo. El control es llevado a cabo en el espacio $\mathbb{R}^2$ (control 2D). Este control está ilustrado en la figura 2.12.b. Queremos que la cámara vea algo similar a lo que se puede ver en la figura 2.13. En la imagen de la izquierda, tenemos la vista inicial, donde el objeto está alejado y descentrado; y en la derecha tenemos la imagen final, donde nos hemos acercado y el objeto es perpendicular a la cámara. El problema de control puede ser expresado en términos de las coor-

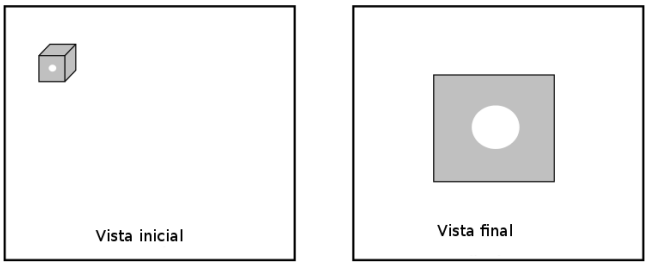

Figura 2.13: Imagen inicial y final

denadas de la imagen (figura 2.14), donde usaremos las esquinas de una cara de un cubo. La vista desde la posición inicial de la cámara está mostrada en rojo y puede apreciarse claramente que está en una posición oblicua respecto al objeto y sus features de interés. La vista deseada está mostrada en azul, donde la cámara está más lejos del objeto y donde su eje óptico es normal al plano del mismo. Skaar [26] propone que muchas tareas del mundo real pueden ser descritas como una o más tareas en el espacio de la cámara, por ejemplo alinear señales visuales en la imagen. La tarea es mover los puntos marcados con círculos a los puntos marcados con rombos. Mover estos puntos en la imagen implícitamente cambia la posición de la cámara y con ella la del robot, como se muestra en la figura 2.14.

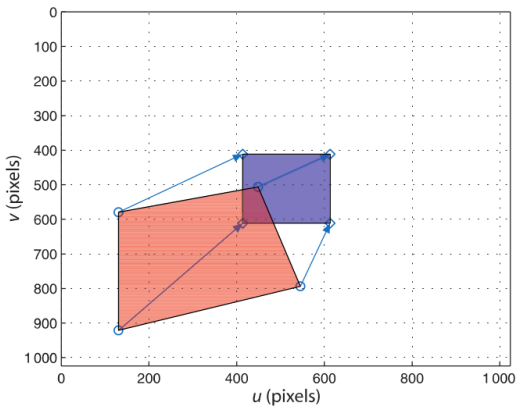

Figura 2.14: Imagen deseada e imagen actual

La posición deseada de la cámara con respecto al objeto está definida implícitamente por el valor de las características (features) de la imagen en la posición objetivo. IBVS presenta un desafiante problema de control debido a que las características de la imagen son funciones altamente no lineales con la posición de la cámara. Por ejemplo, una rotación de la cámara puede causar que las features sufran movimiento horizontal y vertical en el plano de la imagen. Esto se linealiza mediante la matriz Jacobiana de la imagen, que

relaciona el ratio de cambio en el espacio con el ratio de cambio en el espacio de la imagen. Como hemos dicho, al usar matrices Jacobianas, necesitamos información de profundidad entre la cámara y los features, así podemos decir que la Jacobiana se calcula en función de las coordenadas de los features en el plano de la imagen más la profundidad de estos. Hashimoto estima la profundidad en base al análisis de las features [27]. Rives establece la profundidad deseada mejor que actualizarla o estimarla a cada iteración [28]. Este tipo de control debe mejorar el retardo computacional con respecto a PBVS, elimina la necesidad de interpretar la imagen y elimina errores en el modelado de sensores y la calibración de la cámara.

Los tipos de control citados anteriormente son llamados sistemas de control 3D para el control basado en posición (PBVS) y 2D para el control basado en imagen (IBVS). Hoy día se está avanzando constantemente en esta materia y ya están surgiendo sistemas de control híbridos entre PBVS y IBVS llamados 2,5D, en los que parte del error se calcula en 3D y otra parte en 2D. También ha surgido otro tipo de sistemas llamados Motion-Based en los que el error se calcula con respecto al flujo óptico de la imagen, comparado con respecto a uno de referencia.

Problemas de puesta en práctica

Los estándares de vídeo son un problema ya que éstos imponen el uso de vídeo entrelazado para que el ojo humano no tenga la sensación de parpadeo. Una señal de vídeo entrelazado está compuesta por pares de campos secuenciales con la mitad de la resolución vertical que están entrelazados entre sí. Cámaras de alta frecuencia y no entrelazadas están disponibles, pero son muy caras, además de requerir hardware especial.

Las lentes pueden introducir un variado número de distorsiones geométricas. La más significativa es la distorsión radial, en la que los puntos son desplazados a lo largo de lineas radiales en la imagen.

Este efecto es peor cuanto más cerca estemos de los bordes de la imagen. Esto se puede corregir con métodos fotogramétricos. Pero en general Image-Based Visual Servoing es bastante robusto ante la presencia de distorsión en la lente. Las ventajas de tener la cámara montada en el efector final con respecto a tener la cámara fija se encuentran en que se evita la oclusión, se reduce la ambigüedad y se incrementa la precisión.

El procesado de la imagen y la extracción de features debe ser lo más rápido posible. La extracción de features es una parte muy importante de cualquier sistema Visual Servoing. Generalmente se usan como features los centroides de regiones. El tiempo para la extracción se puede reducir en gran parte si solo se analiza una parte de la imagen en vez de toda entera, cuya localización se puede predecir de los cálculos de los centroides previos.

2.4.1. Teoría Image-Based Visual Servoing

En esta parte del capítulo vamos a desarrollar la teoría que abarca este proyecto. La primera parte del estudio nos lleva a un acercamiento al mundo de las cámaras para poder comprender un poco cómo funciona la proyección de una imagen del mundo real (3D) en una imagen en la cámara (2D) y las relaciones que hay entre ambas. En una segunda parte estudiaremos y demostraremos el funcionamiento del sistema de control Image-Based Visual Servoing. Todo el estudio está basado en el contenido del libro de Peter Corke *Robotics, Vision and Control: Fundamental algorithms in MATLAB* [5].

Proyecciones en las cámaras

El modelo pinhole (figura 2.15) es el más básico en formación de imágenes, en el que se asume que se forman imágenes invertidas perfectas. En él tenemos una cámara sin lente, solo con un pequeño orifico por donde pasa la luz. Cuanto más pequeño sea el orificio,

más nítida será la imagen, pero mayor tiempo de exposición se necesitará para su formación, ya que al ser más pequeño pasa menos luz.

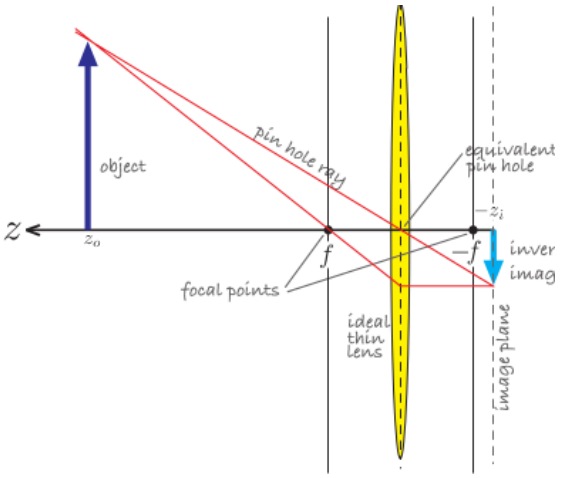

Figura 2.15: Formación geométrica de la imagen para una lente convexa. Sección 2D

Una cámara digital es similar en principio. Tenemos una lente que forma una imagen en la superficie de un chip semiconductor con un array de dispositivos sensibles a la luz que convierten la luz en una imagen digital. El problema del modelo pinhole es que produce imágenes muy tenues, al tener una apertura de lente muy pequeña. La clave para tener imágenes mas brillantes es recoger la luz sobre un área mayor con una lente. Una lente convexa puede formar imágenes igual, pero además permite un mayor paso de luz, al tener mayor diámetro.

Modelo de proyección central

Vamos a pasar a explicar el modelo de proyección central de las cámaras, el cual se puede apreciar en la figura 2.16. Es el modelo que se usa comúnmente en visión por computador, y es el modelo que usaremos en la teoría. Se trata del modelo en el que todos los rayos que pasan por la lente de la cámara convergen en el origen de la

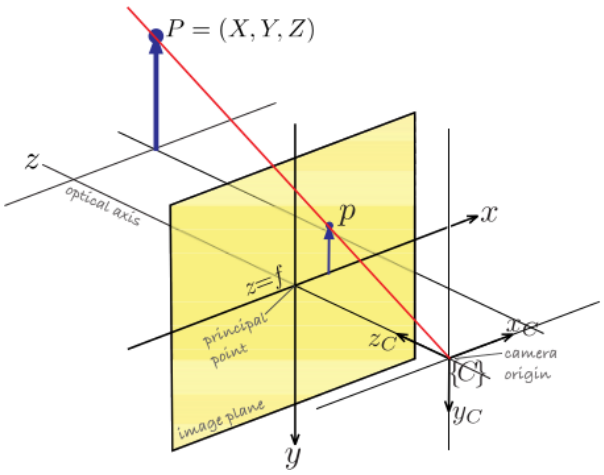

Figura 2.16: Modelo de proyección central de las cámaras

misma {C}. El plano de la imagen se encuentra a una distancia $z = f$ del origen, distancia que llamaremos distancia focal, y está enfrente del mismo, de tal manera que se producirá una imagen no invertida. Considerar que tenemos un punto P, que posee unas coordenadas en el mundo $P = (X, Y, Z)$, este es proyectado en el plano de la imagen como $p = (x, y)$, mediante las siguientes ecuaciones:

$$x = f\frac{X}{Z};\ y = f\frac{Y}{Z}; \tag{2.1}$$

Lo cual es una transformación proyectiva, o más específicamente una proyección en perspectiva desde el mundo hacia el plano de la imagen. Contará con unas determinadas características derivadas de pasar del espacio 3D al 2D.

- Se lleva a cabo un mapeado desde el mundo tridimensional hasta el mundo bidimensional del plano de la imagen.
- Líneas rectas en el mundo son proyectadas como líneas rectas en la imagen del plano.
- Líneas paralelas en el mundo al ser proyectadas se cortan en un punto de fuga, con excepción de las paralelas al plano de la imagen.

- Cónicas en el mundo se proyectan como cónicas en la imagen. Un círculo se proyectará como círculo o como elipse.
- No existe una inversa única, a un determinado punto $p = (x, y)$ no le corresponde un único punto $P = (X, Y, Z)$.
- No se conservan las formas de los objetos, debido a que no se conservan los ángulos internos.

Podemos escribir el punto del plano de la imagen en forma homogénea de la siguiente manera $\widetilde{p} = (x', y', z')$ donde:

$$x' = fX;\ y' = fY;\ z' = Z; \tag{2.2}$$

Podemos escribirlo igualmente de manera homogénea, pero en forma de matriz, que quedaría de la siguiente forma.

$$\widetilde{p} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{2.3}$$

En donde las coordenadas no homogéneas del plano de la imagen quedan:

$$x = \frac{x'}{z'};\ y = \frac{y'}{z'}; \tag{2.4}$$

Si escribimos las coordenadas del mundo en forma homogénea también, quedarán de la forma ${}^{C}\widetilde{P} = (X, Y, Z, 1)^T$, entonces la proyección perspectiva puede ser escrita en forma lineal como:

$$\widetilde{p} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} {}^{C}\widetilde{P} \tag{2.5}$$

Que escrito en forma compacta queda:

$$\widetilde{p} = C\,{}^{C}\widetilde{P} \tag{2.6}$$

C tiene el nombre de matriz de la cámara, y es una matriz de dimensión 3 x 4. ${}^{C}\tilde{P}$ lleva el superíndice C para indicar que son las coordenadas del punto con respecto al sistema de coordenadas de la cámara. La tilde $\sim$ indica que son cantidades homogéneas. En general una cámara tendrá una posición por el mundo ξ_C con respecto al sistema de coordenadas del mundo, como podemos ver en la figura 2.17. La posición del punto P de la figura 2.17 con respecto

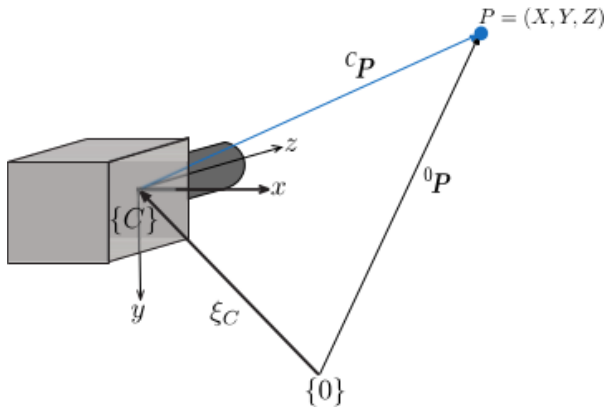

Figura 2.17: Sistemas de coordenadas de la cámara

a la cámara es:

$$ {}^{C}P = (\ominus \xi_C).{}^{0}P \tag{2.7} $$

Que podemos escribir en coordenadas homogéneas como:

$$ {}^{C}P = (T_C^{-1})\,{}^{0}P \tag{2.8} $$

En una cámara digital el plano de la imagen es una cuadrícula de WxH elementos sensibles a la luz que corresponden directamente a los elementos de la imagen o píxeles. Las coordenadas de los píxeles son 2 vectores (u, v) de elementos enteros no negativos. Por convención el origen se encuentra en la esquina superior izquierda. Los píxeles son uniformes en tamaño y están centrados en una cuadrícula regular, con lo que las coordenadas de un píxel están relacionadas

con el plano de la imagen de la siguiente manera:

$$u = \frac{x}{\rho_w} + u_0;\ v = \frac{y}{\rho_h} + v_0; \tag{2.9}$$

Donde ρ_w y ρ_h son el ancho y el alto de cada píxel respectivamente, y (u_0, v_0) son el punto principal, es decir, la coordenada del punto donde el eje óptico interseccionа con el plano de la imagen (figura 2.18).

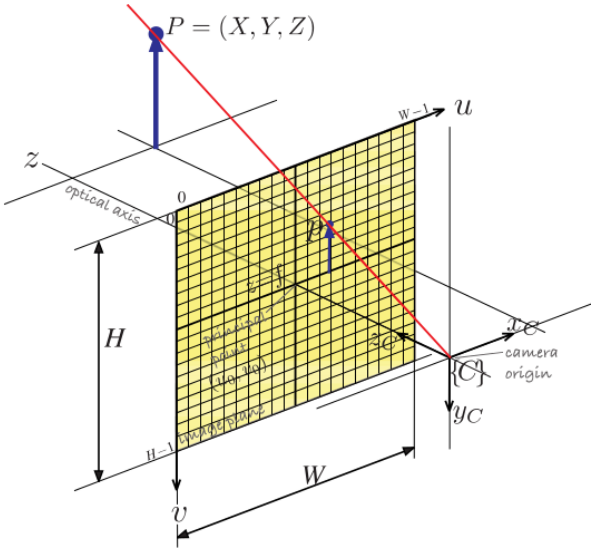

Figura 2.18: Modelo de proyección central, mostrando píxeles del plano de la imagen

Podemos escribir la ecuación Eq. 2.5 para las coordenadas en píxeles anteponiendo una matriz de parámetros de la cámara K.

$$\widetilde{p} = \begin{pmatrix} \frac{1}{\rho_w} & 0 & u_0 \\ 0 & \frac{1}{\rho_h} & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} {}^C\widetilde{P} \tag{2.10}$$

Donde $\widetilde{p} = (u', v', w')$ son las coordenadas homogéneas de un punto P en píxeles. Las coordenadas no homogéneas en píxeles del

plano de la imagen son:

$$u = \frac{u'}{w'};\ v = \frac{v'}{w'}; \tag{2.11}$$

Combinando las ecuaciones Eq. 2.8 y Eq. 2.10 podemos escribir la proyección de la cámara en forma general, donde todos los parámetros se introducen en la matriz de la cámara C.

$$\widetilde{p} = \begin{pmatrix} \frac{f}{\rho_w} & 0 & u_0 \\ 0 & \frac{f}{\rho_h} & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} ({}^0\mathrm{T}_C)^{-1} \widetilde{P} \tag{2.12}$$

$$\widetilde{p} = K P_0 \, ({}^0\mathrm{T}_C^{-1}) \widetilde{P} \tag{2.13}$$

$$\widetilde{p} = C \widetilde{P} \tag{2.14}$$

C es una matriz de 3 x 4 que representa una transformación homogénea que lleva a cabo traslación, escalado y proyección perspectiva. Es habitualmente llamada matriz de proyección o matriz de calibración de la cámara. La proyección puede ser expresada también en forma funcional.

$$p = \mathcal{P}(\mathbf{P}, \mathbf{K}, \xi_C) \tag{2.15}$$

Donde $\mathbf{P}$ es el punto del sistema del mundo, $\mathbf{K}$ es la matriz de parámetros de la cámara y ξ_C es la posición de la cámara. Los parámetros intrínsecos son características innatas de la cámara y del sensor y comprende a f, ρ_w, ρ_h, u_0 y v_0. Los parámetros extrínsecos describen la posición de la cámara y comprenden a un mínimo de seis parámetros para describir traslación y rotación en el espacio $SE(3)$ (Special Euclidean group(3)).

Movimiento de los puntos en la cámara

Esta parte del estudio se basa en lo mostrado en el estudio de las proyecciones de las cámaras. Partiendo de las ecuaciones de-

mostradas anteriormente vamos a empezar a ver el funcionamiento en el que se basa el sistema IBVS. Algo importante a tener en cuenta, es que ahora los puntos de los que tomamos las imágenes y respecto a los cuales hagamos cálculos, serán nuestros puntos de interés, y representarán los objetos que buscamos, son nuestros features, es decir, presuponemos que ya hemos hecho un tratamiento de imágenes y nos interesará guiarnos a ellos. La ecuación Eq. 2.15 a la que hemos llegado antes, es nuestro punto de partida. Si tomamos su derivada respecto de la posición de la cámara ξ_C, nos queda:

$$\dot{p} = J_p(\mathbf{P}, \mathbf{K}, \xi_C)\boldsymbol{\nu} \tag{2.16}$$

Donde $\boldsymbol{\nu} = (v_x, v_y, v_z, w_x, w_y, w_z) \in \mathbb{R}^6$ es la velocidad espacial de la cámara. J_p es un objeto Jacobiano, aunque como hemos tomado la derivada con respecto a una posición $\xi_C \in SE(3)$ en vez de tomarla con respecto a un vector, técnicamente es llamada matriz de interacción. Sin embargo, en el mundo del Visual Servoing es conocida como la Jacobiana de la imagen, o matriz de sensibilidad característica.

Si tenemos una cámara moviéndose con una velocidad propia $\nu = (v, w)$ en el sistema de coordenadas del mundo y que observa un punto con coordenadas relativas a la cámara $P = (X, Y, Z)$. La velocidad del punto con respecto al sistema de coordenadas de la cámara es:

$$\dot{P} = (-w \times P) - v \tag{2.17}$$

Que escrito en forma matricial queda:

$$\begin{aligned} \dot{X} &= Yw_z - Zw_y - v_x \\ \dot{Y} &= Zw_x - Xw_z - v_y \\ \dot{Z} &= Xw_y - Yw_x - v_z \end{aligned} \tag{2.18}$$

Por otro lado, la ecuación de la proyección perspectiva Eq. 2.1

escrita para coordenadas normalizadas queda:

$$x = \frac{X}{Z};\ y = \frac{Y}{Z}; \tag{2.19}$$

Si queremos hacer su derivada temporal, tenemos que aplicar la regla de la cadena, resultando en:

$$\dot{x} = \frac{\dot{X}Z - \dot{Z}X}{Z^2};\ \dot{y} = \frac{\dot{Y}Z - \dot{Z}Y}{Z^2}; \tag{2.20}$$

Partiendo de este resultado, lo introducimos en la ecuación Eq. 2.18, teniendo en cuenta que $X = xZ$; $Y = yZ$; y escribiendo el resultado en forma matricial, nos queda:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} -\frac{1}{Z} & 0 & \frac{x}{Z} & xy & -(1+x^2) & y \\ 0 & -\frac{1}{Z} & \frac{y}{Z} & 1+y^2 & -xy & -x \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ w_x \\ w_y \\ w_z \end{pmatrix} \tag{2.21}$$

Esta ecuación relaciona la velocidad de la cámara con la velocidad del punto del plano de la imagen que estamos viendo, que será el feature marcado. Es decir, tenemos la relación entre la velocidad de la cámara y la del feature. Las coordenadas normalizadas del plano de la imagen están relacionadas con las coordenadas en píxeles por la ecuación Eq. 2.10.

$$u = \frac{f}{\rho_u}x + u_0;\ v = \frac{f}{\rho_v}y + v_0; \tag{2.22}$$

Que podemos reescribir como:

$$x = \frac{\rho_u}{f}\bar{u};\ y = \frac{\rho_v}{f}\bar{v}; \tag{2.23}$$

Donde $\bar{u} = (u - u_0)$ y $\bar{v} = (v - v_0)$ son las coordenadas en píxeles respecto al punto principal. La derivada temporal de esta expresión es:

$$\dot{x} = \frac{\rho_u}{f}\dot{\bar{u}};\ \dot{y} = \frac{\rho_v}{f}\dot{\bar{v}}; \tag{2.24}$$

Sustituyendo las ecuaciones Eq. 2.23 y Eq. 2.24 en la ecuación Eq. 2.21 nos conduce a lo siguiente.

$$\begin{pmatrix} \dot{\bar{u}} \\ \dot{\bar{v}} \end{pmatrix} = \begin{pmatrix} -\frac{f}{\rho_u Z} & 0 & \frac{\bar{u}}{Z} & \frac{\rho_u \bar{u}\bar{v}}{f} & -\frac{f^2+\rho_u^2\bar{u}^2}{\rho_u f} & \bar{v} \\ 0 & -\frac{f}{\rho_v Z} & \frac{\bar{v}}{Z} & \frac{f^2+\rho_v^2\bar{v}^2}{\rho_v f} & -\frac{\rho_v \bar{u}\bar{v}}{f} & -\bar{u} \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ w_x \\ w_y \\ w_z \end{pmatrix} \quad (2.25)$$

Que está expresado en píxeles con respecto al punto principal. Podemos escribirlo en forma matricial concisa.

$$\dot{p} = J_p \nu \quad (2.26)$$

Donde J_p es la matriz Jacobiana de la imagen para un punto feature y tiene una dimensión de 2 x 6. Podemos considerar el movimiento de dos puntos apilando las Jacobianas.

$$\begin{pmatrix} \dot{u}_1 \\ \dot{v}_1 \\ \dot{u}_2 \\ \dot{u}_2 \end{pmatrix} = \begin{pmatrix} J_{p1} \\ J_{p2} \end{pmatrix} \nu \quad (2.27)$$

Lo que va a resultar en una matriz de 4 x 6. Uno de esos movimientos de la cámara se corresponde con la rotación alrededor de la línea que une los dos puntos.

Para el caso de cuatro puntos.

$$\begin{pmatrix} \dot{u}_1 \\ \dot{v}_1 \\ \dot{u}_2 \\ \dot{v}_2 \\ \dot{u}_3 \\ \dot{v}_3 \\ \dot{u}_4 \\ \dot{v}_4 \end{pmatrix} = \begin{pmatrix} J_{p1} \\ J_{p2} \\ J_{p3} \\ J_{p4} \end{pmatrix} \nu \tag{2.28}$$

La matriz Jacobiana resultante es de dimensión 8 x 6, y será no singular si los puntos son no coincidentes o colineares.

Ecuaciones Image-Based Visual Servoing

En la sección anterior hemos visto como los puntos se mueven en plano de la imagen, a consecuencia de los movimientos de la cámara. Lo que nos interesa es lo contrario. ¿Qué movimiento de la cámara es necesario para mover los puntos en el plano de la imagen (features) a una velocidad concreta? Para el caso de cuatro puntos $\{(u_i, v_i),\ i = 1 \cdots 4\}$, con sus correspondientes velocidades $\{(\dot{u}_i, \dot{v}_i)\}$, vamos a invertir la ecuación Eq. 2.28.

$$\nu = \begin{pmatrix} J_{p1} \\ J_{p2} \\ J_{p3} \\ J_{p4} \end{pmatrix}^{-1} \begin{pmatrix} \dot{u}_1 \\ \dot{v}_1 \\ \dot{u}_2 \\ \dot{v}_2 \\ \dot{u}_3 \\ \dot{v}_3 \\ \dot{u}_4 \\ \dot{v}_4 \end{pmatrix} \tag{2.29}$$

Y ahora la vamos a resolver para la velocidad requerida sobre la cámara. Dada una velocidad para los puntos, podemos calcular la velocidad para la cámara, pero, ¿cómo calculamos la velocidad de los

puntos? La forma más simple es aplicar un controlador proporcional lineal simple.

$$\dot{p}^* = \lambda(p^* - p) \tag{2.30}$$

Donde λ es la ganancia. Este controlador guiará las coordenadas de los puntos en el plano de la imagen p hasta el valor deseado de esos mismos puntos en el plano de imagen p^*. Si juntamos esta última ecuación con la ecuación Eq. 2.29, podemos llegar a:

$$\nu = \lambda \begin{pmatrix} J_{p1} \\ J_{p2} \\ J_{p3} \\ J_{p4} \end{pmatrix}^{-1} (p^* - p) \tag{2.31}$$

Este controlador moverá la cámara hasta que los features de la imagen tengan las coordenadas deseadas en la imagen. Es importante resaltar que en ningún momento se ha requerido la posición 3D de la cámara o del objeto (features), todo se ha calculado en términos de lo que se puede medir en la imagen. Para el caso general en el que $N > 4$ puntos, podemos apilar las Jacobianas y resolver una velocidad para la cámara usando la pseudo-inversa.

$$\nu = \lambda \begin{pmatrix} J_{p1} \\ \vdots \\ J_{pN} \end{pmatrix}^{+} (p^* - p) \tag{2.32}$$

Con este controlador moveremos la cámara de tal forma que los puntos en el plano de la imagen queden en la posición deseada. Hay que tener en cuenta que es posible especificar un set de velocidades para los puntos que sea inconsistente, es decir, no hay un movimiento posible de la cámara que pueda resultar en el movimiento requerido de los puntos en el plano de la imagen. En ese caso la pseudo-inversa encontrará una solución que minimice la norma del error de la velocidad de los puntos. Para $N \geq 3$ la matriz puede quedar mal condicionada si los puntos son casi coincidentes o colineales.

A la hora de funcionar esto se manifiesta de manera que algunos movimientos en la cámara producirán pequeños movimientos en la imagen, es decir, el movimiento tiene una baja perceptibilidad. Una vez llegados aquí nos hacemos la pregunta, ¿Cómo calculamos los valores de p^*? Las coordenadas que queremos que tengan nuestros puntos al final pueden ser encontradas por demostración, moviendo el robot al punto deseado y grabando las coordenadas que vea o simplemente sabiendo las geometrías de los objetos que buscamos y de la caracterización que les demos al marcarlos en la imagen como features.

2.4.2. Ejemplos

Vamos a ver algunos ejemplos de sistemas de control Visual Servoing aplicados a muy diferentes áreas de investigación. Mediante estos ejemplos queda patente que el campo de aplicación es tan amplio como se pueda imaginar.

AURORA

Este tipo de control encaja bastante bien en la filosofía de los UAVs. Dentro de este tipo de vehículos tenemos por ejemplo el proyecto AURORA [29] (Project AURORA: Towards an Autonomous Robotic Airship) que usa técnicas de Visual Servoing para el guiado de un dirigible de 10.5 m de largo, 3 m de diámetro, 34 m cúbicos de volumen, una capacidad de carga de 10 kg y velocidad máxima de 50 km/h. El prototipo se muestra en la figura 2.19. El proyecto está en desarrollo y pretende usar el dirigible para monitorizar el entorno y para misiones aéreas de reconocimiento (seguimiento de patrones). Para este tipo de tareas resulta muy útil tener un guiado automático mediante visión. Las tareas de sobrevuelo y seguimiento de lineas se encuentran en mejora constante. Los autores tratan el vuelo, infraestructuras terrestres de hardware y software, modelado dinámico y simulación del vuelo, control y métodos de guiado, estrategias de guiado por control visual, cooperación robótica aire-tierra,reconocimiento dinámico de objetivos y una arquitectura

Figura 2.19: Dirigible proyecto AURORA

híbrida de software dirigible-robot. También presentan resultados satisfactorios del vuelo autónomo del dirigible a través de un set de puntos de paso predefinidos.

Área médica

Otra gran área donde se han introducido sistemas Visual Servoing es en el apartado médico, mediante robots que sirvan para poder realizar operaciones a distancia con gran precisión. La investigación llevada a cabo por Alexandre Krupa [30], presenta un trabajo donde un sistema de visión recupera y posiciona instrumentos quirúrgicos automáticamente durante operaciones laparoscópicas. El instrumento para operar está montado en el efector final de un robot quirúrgico controlado por visión (figura 2.20). El objeti-

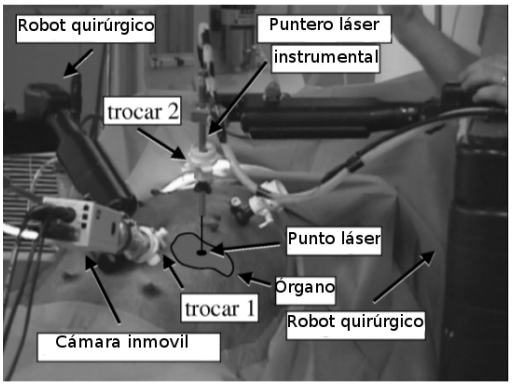

Figura 2.20: Robot quirúrgico de pruebas

vo de la tarea automatizada es llevar con seguridad el instrumento a una localización tridimensional deseada a partir de una posición desconocida u oculta. Diodos emisores de luz se unen en la punta del instrumento, y un soporte de instrumento específico equipado con fibra óptica se utiliza para proyectar puntos del láser en la superficie de los órganos. Estos marcadores ópticos se detectan en la imagen endoscópica y permiten la localización del instrumento con respecto a la escena. El instrumento se recupera y se centra en el plano de la imagen por medio de un algoritmo de control visual que usa como control los errores en las features. Con este sistema, el cirujano puede especificar una posición relativa deseada entre el instrumento y el órgano apuntado. La relación entre la velocidad del instrumento quirúrgico y la velocidad de los marcadores en la imagen se calcula online, por razones de seguridad, se propone un esquema de sistema Visual Servoing de etapas múltiples.

Capítulo 3

Desarrollo del módulo de Visual Servoing

Ahora vamos a estudiar la estructura que posee el módulo de visión que hemos preparado. Todo el proyecto está realizado en base al libro de Peter Corke ya mencionado [5] (figura 3.1). Dicho libro viene acompañado de un sistema de programas o toolbox en MATLAB. Según se van explicando las materias, el libro va indicando comandos para interactuar con los programas y así poder comprender mejor el funcionamiento.

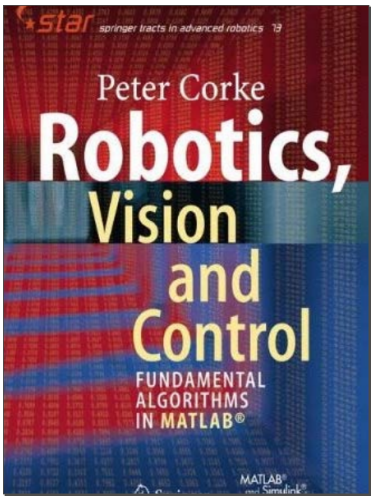

Figura 3.1: Robotics, Vision and Control: Fundamental Algorithms in MATLAB

El libro en sí trata de dos toolbox muy completas; una de robótica en general, que no nos interesa para nuestro tema concreto, y otra toolbox de visión, que es la que nos interesa. Para instruir correctamente el marco teórico desde la formación de imágenes hasta llegar a Visual Servoing, el libro realiza un explicación de toda la teoría necesaria, que se explica en los siguientes pasos.

Primero se explica la manera en que la luz y el color inciden en los ojos, y de que manera se forman las imágenes en el cerebro. Cuando esto se descubrió, se empezaron a producir sistemas de visión por computador que imitaban al humano. También se explican los distintos sistemas de representación de imágenes. Una vez se conoce esto, se pasa a explicar el procesamiento de imágenes y extracción de características de las mismas, que algoritmos son los más conocidos y los que implementa la toolbox. También se explica el uso de múltiples imágenes para una misma escena, y qué beneficios se saca de ello, como puede ser obtener la profundidad de los objetos presentes. Finalmente hay dos temas de Visual Servoing en los que se tratan los dos tipos básicos de Visual Servoing: Position-Based Visual Servoing y Image-Based Visual Servoing. Luego se presentan ejemplos aplicados a robots concretos. Los temas de la toolbox de visión están desarrollados en un orden determinado, de tal manera que al final se tengan todos los conocimientos necesarios para llevar a cabo un sistema de Visual Servoing. Todo el sistema de investigación va a ser montado en el sistema operativo Ubuntu 12.04 32 bit. Vamos a montar un sistema **Image-Based Visual Servoing o IBVS**.

3.1. Concepto organizativo

En esta sección se pretende explicar de qué forma está estructurado el sistema de Visual Servoing. Debido a que un sistema de Visual Servoing no es muy intuitivo de primeras, y como nos basamos en el trabajo de Peter Corke, primero hay que usar y comprender el

funcionamiento de la clase de MATLAB que tiene preparada para tal función. Esto es importante, porque proporciona un orden a la hora de diseñar un sistema propio.

Lo primero que hay que destacar es que vamos a realizar un sistema de Visual Servoing basado en imágenes y no en posición. Este tipo de sistema frente al basado en posición es un poco más sencillo al omitir la estimación de la posición de las features en el espacio. Por lo tanto puede ser más rápido, ya que los algoritmos de estimación de posición consumen bastantes recursos. La clase que proporciona el libro es la clase IBVS.m, la cual, para poder utilizarla, la tendremos que añadir al path de MATLAB por estar trabajando en Linux (si estuviéramos trabajando en Windows tendríamos que añadirla al fichero startup.m).

3.1.1. Clase IBVS

Esta clase de MATLAB es la base de nuestro estudio. Se encuentra en la carpeta de visión y dentro de ella en la subcarpeta examples. La clase IBVS pertenece a la clase base VisualServo, que es la que depende directamente de la clase básica handle de MATLAB. Lo primero que hay que destacar es que para poder lanzar una simulación con esta clase siempre se necesitará por lo menos una variable de entrada de tipo cámara, que es otra clase creada por Peter Corke. Necesitamos por lo menos este tipo de variable de entrada, porque como se vio en la teoría, a la hora de calcular la Jacobiana visual se necesitan varios parámetros de la cámara. En concreto, se necesita saber el ancho y alto de cada píxel, la distancia focal, la resolución, y el punto principal de la imagen.

Para poder ejecutar una simulación, también es necesario especificar unas coordenadas del plano de la imagen, que sean las que queremos que tengan las features a la hora de terminar la simulación. Con esto se está marcando implícitamente la posición que queremos que tenga la cámara con respecto al objeto marcado al acabar la

simulación. Sin embargo, si no se especifican, se usan unos por defecto. Lo mismo ocurre con la ganancia. Al ser todo una simulación de MATLAB y no tener un robot como tal, también se tendrá que especificar una posición de la cámara en el espacio, para luego poder moverla a cada iteración. La clase cuenta con varios métodos para realizar diferentes funciones y incluir o no cierta información en la simulación.

- run Realizar la simulación, resultados se guardan en el objeto
- plot_p Dibujar coordenadas de los puntos frente al tiempo de simulación
- plot_vel Dibujar velocidad de la cámara frente al tiempo
- plot_camera Dibujar posición de la cámara frente al tiempo
- plot_jcond Dibujar condición de la Jacobiana frente al tiempo
- plot_z Dibujar profundidad de los puntos frente al tiempo
- plot_error Dibujar error frente al tiempo
- plot_all Dibujar todo lo anterior en ventanas separadas
- char Convertir el objeto a un string preciso
- display Presentar el objeto como un string

El funcionamiento se realiza entorno a cuatro puntos en el espacio que utiliza como features representativas de un supuesto objeto. Una vez empieza la simulación, la cámara está en una posición inicial, y ahí toma una primera foto de los puntos para pasar a calcular el error en coordenadas de la imagen. Resta las coordenadas que está viendo actualmente la cámara menos las coordenadas deseadas, que son las que se quiere que tenga al final de la simulación. Entonces se calcula la Jacobiana de la imagen. Se necesita como entrada la cámara

definida antes de empezar, la profundidad de los puntos y las coordenadas (puntos) actuales, que son sobre los que calcularemos la Jacobiana.

Con esto calculado ya solo queda convertir el error calculado antes a vector columna. Así se cuadran dimensiones y se calcula la velocidad con la ganancia que tendrá definida por defecto o la que se introduzca como parámetro de entrada. Una vez calculada la velocidad, ésta se aplica a la cámara como una velocidad unitaria, con lo que la cámara se moverá a una posición determinada, es decir, se actualiza la posición de la cámara.

Llegados a este punto el bucle se repite. Se realiza la foto de los puntos, se calcula el error y la Jacobiana, luego la velocidad y finalmente se le aplica a la cámara para actualizar su posición. La variable de control del bucle será el propio error que se calcula a cada iteración. Se mide con la norma total del vector error, en concreto se trata de la norma euclídea, con lo que se calculará como la raíz cuadrada de la suma de las componentes del vector al cuadrado.

Cuando la norma euclídea del error sea menor que alguna definida, estaremos lo suficientemente cerca de los puntos definidos en el plano de imagen inicialmente, es decir, de las coordenadas deseadas. Usar el error de las coordenadas de los puntos en el plano de la imagen como variable de control, mide implícitamente la distancia entre la cámara y el objeto caracterizado por las features. Así que cuando la norma del error sea menor que una usada a modo de tolerancia, la posición del robot, será la posición predefinida por las coordenadas deseadas para las features. Esto ocurre así por tener las features fijas en el espacio. Al simular se muestra la cámara moviéndose por el espacio y su vista (figuras 3.2 y 3.3).

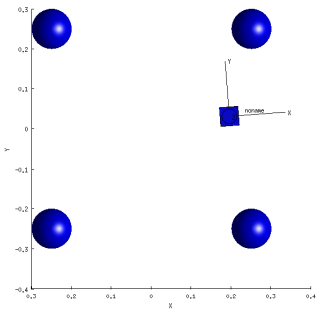

Figura 3.2: Ventana IBVS. Vista de la cámara moviendose

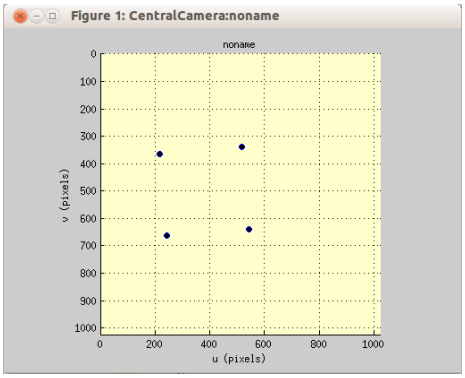

Figura 3.3: Ventana IBVS. Vista de lo que ve la cámara

3.1.2. Sistema de programas desarrollados

Ahora se va a presentar el desarrollo del programa propio principal de Visual Servoing y de todos los subprogramas también desarrollados para poder conseguir una simulación completa. Primero hay que destacar que ahora ya contamos con un robot simulado que porta una cámara y está definido en un entorno de trabajo determinado, todo funcionando en OpenRAVE.

El bucle de control es en esencia el mismo que usa Peter Corke y que hemos explicado antes, pero vamos a tener ciertas diferencias. El de Peter Corke es todo una simulación matemática alejada de la realidad, mientras que en nuestro caso vamos a mover un robot dentro de un entorno. Hay cosas que no necesitamos y otras extra

que necesitaremos realizar por nuestra cuenta. Empezaremos porque ahora no vamos a necesitar una posición en el mundo de la cámara. Nosotros tenemos una montada sobre el robot en su simulador "real" y está en alguna posición concreta. De lo que deducimos que tampoco tendremos que actualizar la posición de la cámara, porque ahora moveremos a un robot con una velocidad y eso en si ya mueve la cámara. Tampoco vamos a necesitar la posición en el mundo de los 4 puntos, ya que estos están en otro sitio del simulador desde donde serán vistos por la cámara del robot.

Sin embargo vamos a necesitar de otras cosas extras. En la clase de Peter Corke los puntos son puntos como tal, mientras que en nuestro simulador tenemos precargadas cuatro esferas. Para poder obtener coordenadas de ellas, necesitamos realizar un tratamiento de imágenes por ordenador. Tendremos que realizar una segmentación de la imagen y un etiquetado de 4 regiones o blobs de interés, que serán nuestras cuatro esferas. Una vez hecho esto, sólo necesitamos el centroide de las cuatro esferas, y éstos serán nuestros puntos para el guiado del robot.

Hay que destacar que a la hora de calcular el error, el cual es parte de la expresión de la velocidad, y donde cerramos el lazo de control, tiene que tener un orden determinado. Fijándonos en los ejemplos que podemos ver en la clase de Peter Corke, se ve que siempre define la matriz de puntos en un orden determinado. Esto hace que al calcular el error, siempre esté calculado en el mismo orden y no estemos mezclando puntos, cosa lógica por otra parte. El orden que vamos a seguir es el siguiente: El primer punto de nuestra matriz de cuatro puntos será el que tiene las coordenadas más arriba y a la izquierda. El segundo punto se corresponde al que tiene debajo (misma coordenada horizontal, pero mayor coordenada vertical). El tercer punto es el que está a la derecha del segundo (misma coordenada vertical, pero mayor coordenada horizontal). El cuarto

y último punto es el que está encima del tercero (misma coordenada horizontal, pero menor coordenada vertical). Para que quede más claro, se muestra dicho orden en la figura 3.4.

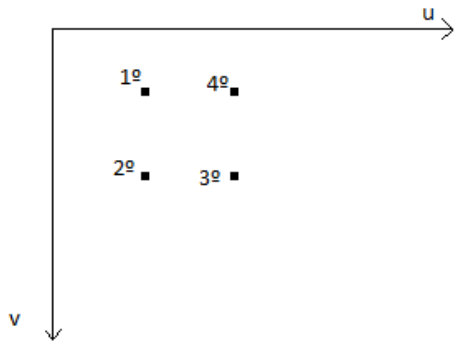

Figura 3.4: Ordenación features de control

Esto resulta en un problema adicional. La clase de Peter Corke al tener simulados los puntos, los tiene siempre en orden, nosotros estamos segmentando de una imagen con 4 esferas verdes. Al usar una segmentación de OpenCV y un posterior etiquetado, este no ocurre siempre en orden y menos aún en el mencionado. Cada vez que en el programa recojamos por los puertos abiertos los datos de la segmentación, tendremos que pasarlos por un programa extra que los ordene siempre en el mismo orden. La función de ordenado sumará las coordenadas x e y de cada uno de los puntos de manera que así obtenemos el 1º y 3º de los puntos. El primero es el que tiene la menor suma de sus coordenadas y el tercero el que posee la mayor, según los ejes mostrados en la imagen anterior. Los otros dos puntos restantes se ordenarán comparando su coordenada horizontal. El 4º punto será el que tenga la mayor coordenada horizontal y el punto que nos queda es el 2º por descarte. Si tenemos en cuenta que el eje u se corresponde al X y el eje v al Y, el eje Z queda hacia dentro del papel. Al pasar el sistema por el programa de ordenado descrito estamos metiendo ciertos límites. Si estamos por ejemplo en dirección perpendicular a los features, si rotamos en Z más de

45º o menos de -45º vamos a cambiar el orden de los puntos y en ese punto el programa puede fallar al tener dos puntos una misma coordenada horizontal.

Flujo de control

El funcionamiento va a ser similar a la clase IBVS en la que se inspira, salvo algunas diferencias que vamos a explicar con detalle a continuación. Básicamente se realizará una primera recogida de datos de la segmentación y etiquetado. Se realizará su ordenación mediante un programa y así se podrá calcular el error en coordenadas de la imagen. Se obtiene la Jacobiana lo que permitirá calcular la primera velocidad y al final se realiza el envío de la misma al robot a través del puerto preparado para ello. En este punto se entra en el bucle de control. Realizamos otra recogida de los datos de la segmentación de la imagen que este viendo actualmente en la cámara, que será ligeramente distinta porque ya se ha movido el robot. Ahora se realiza otra vez una ordenación de la matriz de los puntos. Volvemos a calcular la velocidad y a enviarla. El programa de velocidad nos devolverá el error al programa principal y será el que comparemos con uno preestablecido para ver lo cerca que estamos de las coordenadas deseadas. Repetiremos el proceso hasta que se cumpla la condición del error. Aquí el bucle puede funcionar también hasta llegar a un número de iteraciones preestablecidas. Llegados al punto de salir del bucle ya solo queda cerrar los puertos, y al final se devuelve el error final. También saldrá por pantalla el tiempo total que ha tardado en completarse la acción, con la función tic toc de MATLAB. Este tiempo se devuelve para poder realizar pruebas con las ganancias del sistema y encontrar mejores tiempos de respuesta para el sistema. Tener las mismas coordenadas o casi, una vez acabamos, con respecto a las deseadas, implícitamente obliga a que el robot este en la misma posición que la pregrabada con respecto a las features que representen el objeto en cuestión.

El sistema puede ser lanzado para que funcione con unas iteraciones fijas determinadas por el usuario y así poder realizar comprobaciones del funcionamiento del mismo, mediante la medida del error o la posición de los motores. Por otra parte se puede lanzar para que termine una vez se consiga un error determinado por el usuario. Se incluyen dos diagramas de flujo en la siguiente página para poder comprender de una forma más esquemática el funcionamiento del sistema (figura 3.5 y 3.6).

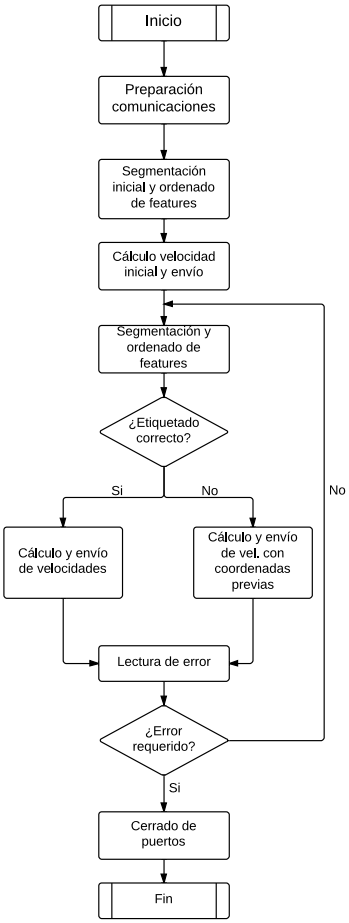

Figura 3.5: Esquema de funcionamiento del sistema

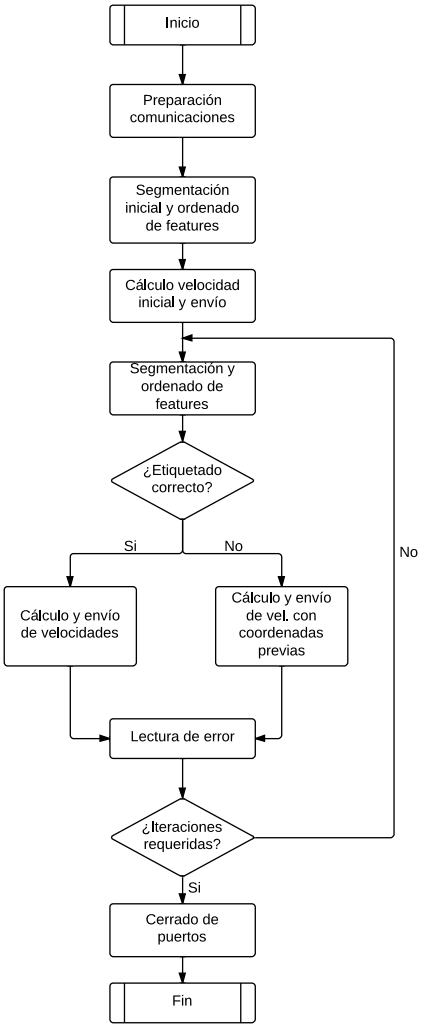

Figura 3.6: Esquema de funcionamiento del sistema

3.2. Implementación

En esta parte del capítulo se va a explicar cual es la infraestructura del sistema de simulación y como está conectado con el sistema de control. Por un lado vamos a tener al robot simulado junto con su cámara y entorno en OpenRAVE, y por otro lado vamos a tener

en MATLAB funcionando los programas de Visual Servoing. Se van a comunicar entre ellos a través de unos puertos que cruzarán datos mediante YARP.

También hay que comentar que al usar OpenCV para segmentación y etiquetado, surgió un problema. El etiquetado, siempre que se estén viendo las cuatro esferas, devuelve cuatro blobs, pero alguna vez dos de los cuatro blobs son el mismo punto repetido dos veces. Con lo que nos está dando las coordenadas de tres de los cuatro puntos. Esto solo ocurre en algunas posiciones determinadas del sistema, y al estar moviéndose el robot, ocurre en muy pocas ocasiones comparado con el total de iteraciones del sistema.

Ante este fallo se pensó en dos formas de atacarlo. Una fue incorporar un bucle infinito en el programa que recibe el etiquetado y que se leyera el puerto hasta que se recibieran correctamente los datos. La otra fue guardarse a cada iteración las coordenadas de la anterior, para que en caso de fallo se aplicaran las coordenadas anteriores. Lo que nos lleva a aplicar la misma velocidad que en el paso anterior. La primera opción parece más inestable ya que puede hacer que alguna iteración tenga un tiempo de cálculo bastante superior al resto, además de que los puertos de YARP son algo delicados de tratar y pueden bloquearse. Se optó por la segunda opción, ya que si aplicamos dos veces la misma velocidad, en pocas iteraciones nos dará un etiquetado correcto porque esto solo ocurre alrededor de determinadas posiciones. Además no aumentamos el tiempo de proceso del bucle y se mantiene una trayectoria lineal.

Para la comprobación de que se están recibiendo correctamente las cuatro etiquetas o blobs, se realizó un pequeño programa que calcula las distancias euclídeas entre los puntos. Se comprobó que cuando fallaba, 2 de las 12 distancias son 0 al estar repetidos 2 puntos de los 4. Después de todo lo comentado, el sistema completo va a contar con los siguientes programas (figura 3.7), que se explicarán

a continuación.

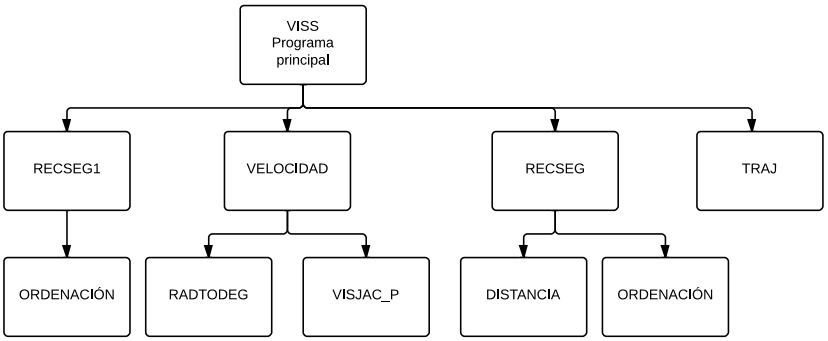

Figura 3.7: Sistema de programas de control

3.2.1. Programa viss

Viss (Visual Servoing) es el programa principal de control y desde el que se realizan las llamadas a los demás programas. Nuestro programa principal se puede resumir con el empiece del programa y precarga de YARP en MATLAB, abrir los puertos entre MATLAB y OpenRAVE, que son tres puertos: uno para el envío de velocidades, y otros dos para recibir los datos del etiquetado de la segmentación y de la posición de los motores.

El puerto para el envío de velocidades es de tipo yarp.Port. Los otros dos puertos, al ser puertos de lectura y poder quedarse bloqueados, se decidió que fueran de otro tipo, en concreto puertos yarp.BufferedPortBottle. Los puertos se abren por el lado de MATLAB, para justo después conectarlos con los puertos que posee ASIBOT preparados para los mismos fines. Para el envío de velocidades nos conectamos al puerto '/ravebot/cartesianServer/command:i', para la recepción de datos de la segmentación y etiquetado nos conectamos a '/colorSegmentor/state:o'. Y finalmente para la

recepción de datos de posición de los motores nos conectamos al puerto '/ravebot/state:o'.

Después de este paso, se define el modelo de cámara que porta el robot, es decir, se definen los parámetros intrínsecos de la cámara, distancia focal, ancho y alto de píxel, resolución y punto principal. Ahora se define una primera matriz que guarda las posiciones de los motores, y otra que guarda la norma del error de las coordenadas de la imagen. En este punto se realiza la primera lectura de las posiciones y de las coordenadas de las features, que nos devuelve ordenadas su programa, y se guardan en sus respectivas matrices.

Con la primera lectura de las posiciones, se llama al programa que calcula y envía la velocidad. Este nos va a devolver el error en coordenadas de la imagen mediante la resta de las coordenadas que se desean obtener, menos las que se tienen actualmente. Ahora se entra al bucle de control, en donde la variable de salida es el error mencionado. El cual es un error umbral mínimo para poder salir del bucle. Se vuelve a leer la posición de los motores y se guarda en su matriz. Posteriormente se leen y ordenan las coordenadas de las features, para volver a llamar al programa de velocidad, y que este nos devuelva la error de nuevo y se almacene en su matriz.

Una vez se cumple la condición del error requerido se sale del bucle y se cierran los puertos utilizados para las comunicaciones. Ahora el programa busca la iteración en la que norma del error pasó por un valor determinado. También se busca la iteración, si es que la ha habido, en la que la norma del error se repitió un número determinado de iteraciones. Finalmente se dibuja en una ventana la norma del error, y otra se subdivide en dos para mostrar la posición de los motores de traslación por un lado, (movimiento en X, Y, Z) y por otro la posición de los motores de rotación (rotación en X, Y, Z). El diagrama de flujo del programa se muestra en la figura 3.8.

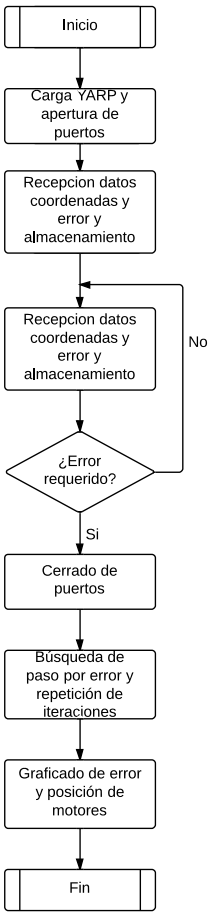

Figura 3.8: Esquema de funcionamiento del programa viss

3.2.2. Programa velocidad

Este programa es el que se encarga del cálculo y envío de la velocidad a cada iteración. El programa recibe entre sus datos de entrada las coordenadas deseadas finales y las coordenadas actuales, que son las que acaba de captar la cámara en esa iteración. Con esto se calcula el error en coordenadas de la imagen. Luego se calcula la Jacobiana visual con este error y con la profundidad de los puntos. Después se mide la norma del error, y si este es grande se aplica una determinada ganancia para los motores de traslación y otra bastante menor para los de rotación. Si la ganancia es menor que cierto valor,

esta se reduce a la mitad.

La Jacobiana visual se calcula a través de un programa de la toolbox de visión de Peter Corke, en concreto Visjac_p, el cual calcula la Jacobiana acorde a como se demostró en la teoría. Finalmente con el error, la ganancia y la Jacobiana visual, se calcula la velocidad y se envía a través de los puertos mediante YARP. Se envían los datos metiéndolos en un Bottle que es una clase especial de YARP, precedido de dos Vocab que indican que enviamos velocidad en coordenadas cartesianas. Se muestra un diagrama de flujo en la figura 3.9.

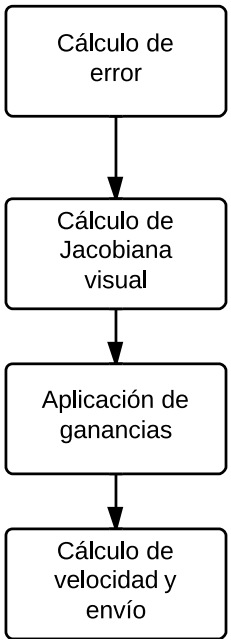

Figura 3.9: Funcionamiento programa de velocidad

La velocidad se calcula como en el sistema de referencia de las cámaras y es como se indica en la figura 3.10.

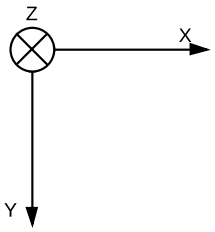

Figura 3.10: Sistema de referencia del sistema de control

3.2.3. Programas recseg, recseg1

Estos son dos versiones del mismo programa, solo que uno realiza la comprobación de que la segmentación es correcta y el otro no. Recseg1 no realiza esa comprobación, ya que es el se ejecuta fuera del bucle, y al ser la primera vez que se llama no tenemos unas coordenadas previas. El programa recibe los datos de OpenRAVE por el puerto preparado a través YARP mediante un Bottle. Se separan en cada uno de los puntos con su coordenada x e y respectivamente, y se montan en una matriz. Aquí Recseg1 llama al programa de ordenación y devuelve la matriz ordenada.

Recseg en vez de ordenar los puntos directamente, primero calcula las distancia euclídeas entre ellos. Se calculan 12 distancias y posteriormente se mira si alguna es cero. Si alguna es cero es porque en esa iteración la etiquetación ha fallado y se está repitiendo algún punto. Si la etiquetación ha fallado, se envían las coordenadas de la iteración anterior, sino se procede a su ordenación y se devuelven al programa principal. Diagrama de flujo en la figura 3.11.

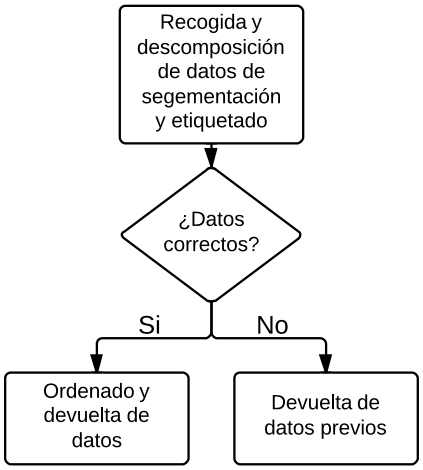

Figura 3.11: Funcionamiento programa de recogida y ordenado de datos

3.2.4. Programas traj, radtodeg, distancia

El programa traj es un simple programa que mediante un Bottle recibe los datos de la posición de los motores del robot, y los va almacenando en una matriz para luego poder tratar estos datos. El programa radtodeg es un simple cálculo para para pasar un ángulo de entrada en radianes y devolverlo en grados. Por último el programa distancia, es otro programa que realiza un cálculo simple de la distancia euclídea entre dos puntos.

3.3. Tutorial

Para poder lanzar el sistema vamos a utilizar por un lado el simulador con su infraestructura, y por el otro el sistema de control en MATLAB. Primero tenemos que lanzar el simulador del robot para lo cual antes tendremos que lanzar en una terminal el servidor YARP

```
[terminal1] yarp server
```

En otra terminal lanzamos la simulación, para lo cual vamos a utilizar el manager de YARP. Primero nos moveremos a la carpeta donde tengamos instalado el simulador de ASIBOT

```
[terminal2] cd asibot/app
[terminal2] gyarpmanager
```

Al teclear estos comandos nos saltará un programa con este aspecto (figura 3.12). En dicho programa pincharemos en la ventana

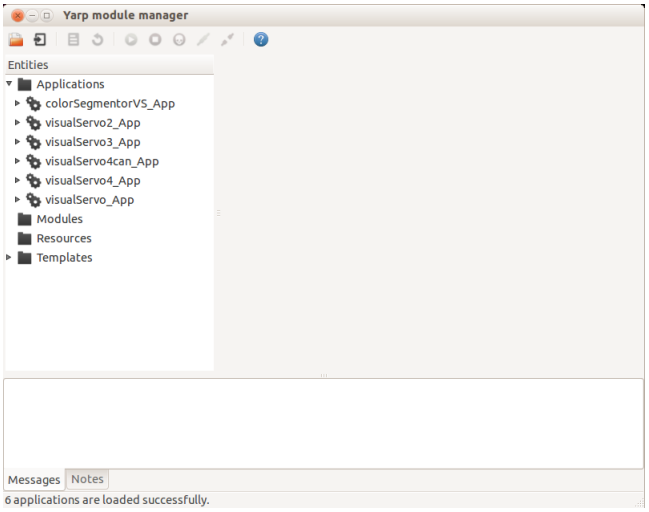

Figura 3.12: Gyarpmanager

Applications. Una vez hecho se nos despliegan 4 opciones, la primera, cuyo nombre es colorSegmentorVS_App es la aplicación con la que lanzaremos el simulador de ASIBOT. Las otras tres constituyen los entornos de pruebas del sistema consistente en un robot cartesiano que se mueve por el espacio. VisualServo_App es el entorno de pruebas pero sólo con 2 grados de libertad, visualServo2_App es el mismo entorno pero con 3 grados de libertad y finalmente visualServo3_App es el mismo entorno pero con 6 grados de libertad. Finalmente los entornos del robot cartesiano con 6 grados de libertad y dentro de la cocina son, visualServo4_App y visualServo4can_App. El primero se guía mediante esferas y el segundo se guía hacia la lata roja. Una vez decidido el sistema que queramos lanzar, pincharemos en el dos

veces y se nos desplegará a la derecha una pestaña de la aplicación (figura 3.13). Ahora pincharemos en el icono de play que contiene la aplicación. Si lanzamos la aplicación con el entorno de ASIBOT nos

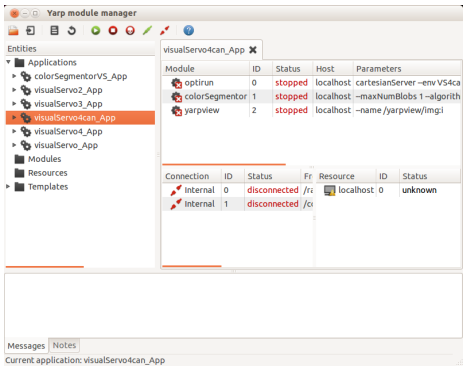

Figura 3.13: Gyarpmanager con aplicación selecionada

saltará una ventana parecida a la figura 3.14. Si lanzamos cualquiera de los entornos de prueba nos saltará una ventana del tipo figura 3.20. Si lanzamos el entorno de la cocina con el robot cartesiano

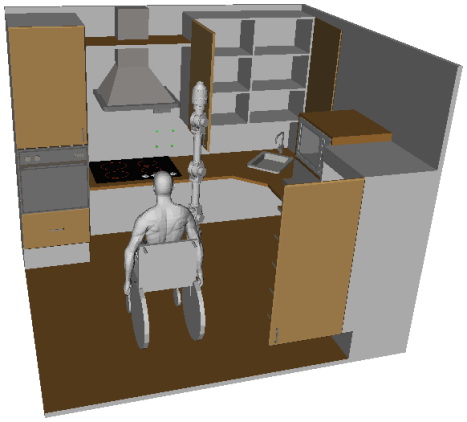

Figura 3.14: Simulador de ASIBOT

y con detección de esferas obtendremos un sistema como la figura 3.15.

Si lanzamos la cocina con el robot cartesiano y detección de la lata obtendremos una ventana similar a la figura 3.16. Una vez lanzado el entorno correspondiente, tendremos dos ventanas, una que

Figura 3.15: Cocina de ASIBOT con robot cartesiano y detección de esferas

Figura 3.16: Cocina de ASIBOT con robot cartesiano y detección de lata

será una de las anteriores y otra más pequeña que corresponde a la vista de la cámara del robot, para activarla tendremos que pulsar en el símbolo de connect links. Una vez pulsado por esa ventana veremos algo similar a la figura 3.17.

Por otro lado abrimos MATLAB. Una vez abierto lo primero será colocarnos en la carpeta donde tengamos los sistemas de programas adaptados a cada entorno de trabajo de los 4 descritos que contiene el manager de YARP. En las últimas versiones los programas para el sistema de control están incluidos en el repositorio de ASIBOT, tendremos que desplazarnos a la siguiente carpeta dentro de MATLAB.

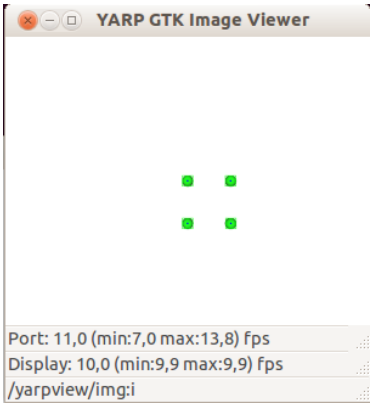

Figura 3.17: Vista de la cámara de la simulación

```
>> cd asibot/main/matlab/visualServo
```

Debido a que se usan algunas partes de la toolbox de visión de Peter Corke (subcarpeta rvctools), estos programas deben incluirse en el path de MATLAB para que sepa a que programas se llaman y no produzca error. Ahora tenemos que definir las coordenadas que deseamos que se vean en la cámara del robot cuando finalice la acción de control. Estas las podemos definir dentro de las posibilidades que ofrece un cuadrado en una resolución de 1280 x 1024. Esta matriz será la matriz de puntos deseados pStar y se corresponde con la primera entrada del sistema.

En el caso de ASIBOT también dependerá de si el robot puede quedar en una posición en la que se vean esas coordenadas para los puntos features. Los puntos feature se tienen que definir en el orden indicado anteriormente (figura 3.4). Dentro de la carpeta anterior tenemos una subcarpeta que define unas matrices de ejemplo para usar como coordinadas finales.

```
>> cd example
>> load matrices
>> cd ..
```

Otra opción para definir los puntos que deseamos ver al final será moviendo el robot manualmente a esa posición deseada, mediante los comandos definidos en el estado del arte. Una vez en esa posición usaremos un programa que recoge y ordena las coordenadas. Dentro de la misma subcarpeta de las matrices de ejemplo, tendremos que teclear en el IDE de MATLAB lo siguiente.

```
>> pStar = recpos();
```

En este punto tenemos una matriz de 2 x 4, en la que cada columna es un punto con su coordenada u (x) y v (y) correspondiente, ordenados de la forma estipulada. Hay que recalcar que para obtener un correcto funcionamiento del sistema las coordenadas deseadas o finales que definamos para el sistema deben de encontrarse centradas en la resolución de la cámara. Una vez definida la posición deseada del robot respecto de los features, tendremos que posicionarlo en otra posición distinta en la que lógicamente las coordenadas de los features en la imagen también son distintos. Ahora para que empiece la acción de control teclearemos lo siguiente en el IDE.

```
>> [a,b,c,d,e] = viss(pStar, profundidad);
```

La primera entrada, “pStar”, correspondiente a la matriz de puntos de referencia, ya hemos explicado como obtenerla. La segunda entrada del sistema, “profundidad” deberá ser un número o vector correspondiente a la profundidad de los puntos respecto de la cámara. El programa tiene una última entrada opcional, para indicar el modo de funcionamiento del mismo. Podemos hacer que funcione con un número de iteraciones fijas o que funcione hasta llegar a un error determinado. Si no introducimos la tercera entrada, el sistema funcionará por error, si introducimos un número como tercera entrada, el sistema funcionará con ese número como número de iteraciones. De tal manera que teclearemos lo siguiente.

```
>> [a,b,c,d,e] = viss(pStar, depth, nºiter);
```

Si queremos por ejemplo realizar una simulación por error podemos teclear lo siguiente.

```
>> viss(pStar1, 2);
```

Si queremos realizar una simulación con un número de iteraciones fijas podemos teclear lo siguiente.

```
>>[a,b,c,d,e] = viss(pStar1, 2, 2000);
```

De los cinco valores de retorno, el primero, "a", corresponde al error en x e y de cada punto al finalizar la acción de control. El segundo valor de retorno, "b", corresponde a una matriz con las posiciones de los motores a cada iteración. El tercer valor de retorno, "c" se corresponde con un vector que contiene la norma del error euclídea de las coordenadas de la imagen con respecto a las deseadas a cada iteración. El cuarto valor de retorno, "d", corresponde a la iteración en la que se repite un mismo valor de error durante 100 iteraciones. El quinto y último valor de retorno, "e", corresponde a la iteración en la que la norma del error pasó por el valor de 5. Los dos últimos valores de retorno están pensados para cuando el sistema funciona por iteraciones y ver como se comporta a la larga. De todas formas si no existe un valor de iteración en el que el error se repita 100 veces nos devolverá la última iteración. Si el sistema funciona por error y este acaba cuando la norma del error es menor que 5, el último valor de retorno también será la última iteración.

Por el IDE de MATLAB, al finalizar el bucle, veremos el tiempo total empleado para llevar al robot a la posición deseada, además de un mensaje de que el sistema ha finalizado por error o por iteraciones. Finalmente se lanzarán dos ventanas gráficas de MATLAB con las gráficas de la norma del error a cada iteración y de la posi-

ción de los motores de traslación y de rotación (figuras 3.18 y 3.19).

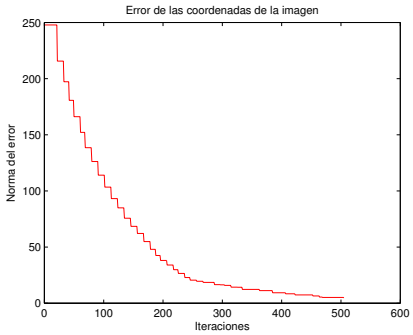

Figura 3.18: Gráfica de los motores de traslación

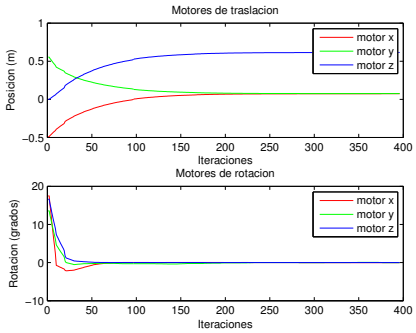

Figura 3.19: Gráfica de los motores de rotación

Una vez finalizado, en el manager de YARP pulsaremos el botón de Stop Applications para cerrar la simulación, y cerraremos las terminales y MATLAB. Dentro del funcionamiento del sistema hay que comentar lo más importante, y es que para que el sistema de Visual Servoing funcione, como su nombre indica, la cámara tiene que estar viendo siempre las features que utilicemos para el guiado. En nuestro caso pueden ser las 4 esferas o la lata de coca cola. En el caso de que se deje de ver una o más esferas, o la lata, saltará un WARNING y el programa se parará por fallo en el mismo. Si queremos probar con el simulador de ASIBOT aparte de lanzarlo en el manager de YARP como se ha indicado antes, una vez se tienen las

coordenadas deseadas y se ha posicionado al robot en una posición distinta, hay que teclear lo siguiente en otra terminal.

```
[terminal]  yarp rpc /ravebot/cartesianServer/rpc:i;
[terminal] tool 1
```

En MATLAB el sistema funciona de manera más simple.

```
>> a = viss2(pStar);
```

El sistema de ASIBOT no llega a funcionar bien y se tendría que terminar de desarrollar el solver de cinemática del robot para conseguir un funcionamiento más acorde.

Si se desean realizar cambios en los entornos se tiene que acceder a la carpeta asibot/app/ravebot/models y ahí se encuentran los 6 entornos descritos en archivos XML. Para cambiar las propiedades de las esferas, por ejemplo, habría que ir a la parte del XML de cualquiera de los entornos siguiente:

```
<!-- Introducción de esferas-->
  <KinBody name="boxa">
  <Translation>-1.55 -0.95 -0.465</Translation>
    <!-- floor should never move, so make it static-->
    <Body type="static">
      <Geom type="sphere">
        <!--extents>1 1 1</extents-->
        <diffuseColor>0 1 0</diffuseColor>
        <radius>0.02</radius>
        <ambientColor>0.6 0.6 0.6</ambientColor>
      </Geom>
    </Body>
  </KinBody>
```

Donde está definida la esfera “boxa” y se encuentra situada en el punto del espacio definido por Traslation y podemos cambiar su tamaño o color con las opciones radius y diffuseColor.

3.4. Consideraciones prácticas

Esta parte trata de como se ha evolucionado, desde no saber nada de un sistema de control por visión, hasta implementar el sistema completo. Posteriormente se explica la forma de usar el sistema desarrollado. Lo primero que se ha realizado es una lectura del libro de Peter Corke [5]. Después se estudió más en profundidad la parte de Visual Servoing, realizando pruebas con la clase de Image-Based Visual Servoing IBVS.m, hasta desarrollar un programa simulado desde las funciones que facilita el libro.

Una vez con el concepto más claro, se instaló toda la infraestructura de ASIBOT dentro Ubuntu y se comenzó a probar su funcionamiento básico. Llegados a este punto y después de muchas pruebas, se comprobó que ASIBOT no estaba preparado para recibir los datos que proporciona el sistema de Visual Servoing. A cada iteración se obtienen velocidades en coordenadas cartesianas del sistema de referencia de la propia cámara, es decir, se obtiene un vector de la forma $v = (v_x, v_y, v_z, w_x, w_y, w_z)$. ASIBOT puede recibir velocidades cartesianas, pero referidas a la base del robot. Se intentó desarrollar un solver de cinemática que solucionara el problema, sin tener éxito en tal cuestión. Aquí se decidió montar otra plataforma de pruebas sobre OpenRAVE mientras se preparaba a ASIBOT. Dicha plataforma de pruebas consiste en un entorno vacío donde se tiene un cubo con una cámara y cuatro esferas para hacer de features. Dicha plataforma se puede ver en la figura 3.20. Con la plataforma ya preparada para probar primero hay que resaltar unos apuntes sobre la cámara y el modelado de esta, que se hicieron notar a la hora de montar y probar el sistema.

Cámara

Hay que realizar una mención especial a la parte de la cámara, ya que esta tiene que estar modelada con sus parámetros intrínsecos tanto en MATLAB, como en el fichero XML que compone el en-

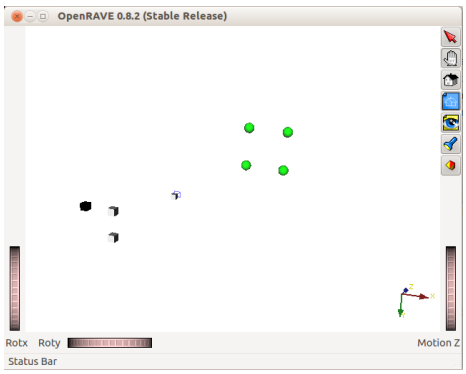

Figura 3.20: Plataforma de pruebas para sistema de IBVS

torno de pruebas de OpenRAVE. Tendremos que tener el mismo modelado en ambos sitios. La cámara la hemos modelado como una cámara ip Axis 207 MW, que es una cámara que posee la universidad para este tema. De la web del fabricante se puede encontrar un pdf de donde sacamos que la cámara posee una distancia focal de 3,6 mm, un tamaño de pixel de $3.6\mu m$ tanto en x como en y, y una resolución de 1280 x 640 píxeles. En MATLAB la modelamos con estos parámetros directamente a través de una clase de la toolbox de visión, en OpenRAVE la modelamos con la matriz de parámetros intrínsecos. En ambos programas se realiza mediante la siguiente aproximación. Se ha usado la documentación *Camera Calibration Toolbox for MATLAB* [31].

$$KK = \begin{pmatrix} \frac{cam.f}{\rho_x} & 0 & u_c \\ 0 & \frac{cam.f}{\rho_y} & v_c \\ 0 & 0 & 1 \end{pmatrix}$$

Siendo $cam.f$ la distancia focal y ρ_x, ρ_y, el tamaño de píxel en x e y respectivamente. Por otra parte u_c y v_c son el punto principal del plano de la imagen en x e y. Teniendo en cuenta los parámetros que hemos sacado de la web del fabricante nuestra matriz resulta ser.

$$KK = \begin{pmatrix} 1000 & 0 & 640 \\ 0 & 1000 & 512 \\ 0 & 0 & 1 \end{pmatrix}$$

En un principio con el esquema comentado de funcionamiento, se realizó un programa que recibía imágenes procedentes de la simulación. Dentro de MATLAB, con las funciones de Peter Corke, se realizaba una segmentación y etiquetado, para posteriormente ordenar los puntos. Esto funcionaba, pero tenía un problema muy grande y era el tiempo que tardaba el bucle de esta manera. El bucle tenía un tiempo de funcionamiento de entre 0,5 a 2 segundos lo cual es un tiempo totalmente fuera de orden para nuestro propósito, aparte de aleatorio según la imagen recibida. Este programa era muy lento por dos motivos. Primero porque se pasaban imágenes enteras desde el simulador a MATLAB. Cada imagen son tres matrices de 1280 x 1024 píxeles. Segundo porque se usaban funciones de la toolbox de un ámbito muy genérico y poco eficientes, que detectaba toda clase de features por la imagen.

Cuando se ven los ejemplos del libro y se prueba la clase IBVS hay que notar que todo lo que se hace y calcula en ella es de manera unitaria, se calcula una velocidad que luego se aplica a la cámara como si fuera un paso. Sin embargo nos enfrentamos a probarlo con una simulación realista o con un robot como tal, y eso no es posible. Nosotros aplicamos una velocidad al robot y esta es constante hasta que el sistema vuelva a procesar otra velocidad y se la envíe.

Esto hace notar que prima la velocidad en el bucle de control. Queremos que los pasos de recibir y ordenar los datos del etiquetado, además de calcular la velocidad y enviarla, tienen que ser lo más rápidos posibles. Debido a esto se pasó a usar el puerto en streaming que usa OpenCV, por lo que se pasó a recibir por MATLAB los

datos de la segmentación directamente. Con esto se consiguió una velocidad más acorde a nuestros requisitos, que está entorno a 0.1 segundos/iteración. Hay que decir que este tiempo está relacionado directamente con la potencia del ordenador en el que se pruebe el sistema. En este caso está montado en un ordenador portátil de gama media.

Con esta infraestructura se probó el sistema con el entorno de pruebas, y ahí se pudo comprobar de forma sencilla su correcto funcionamiento. Con este entorno al ser más sencillo que el entorno de ASIBOT, es donde mejor se puede comprobar el funcionamiento del sistema y de como afecta los parámetros a su comportamiento. Finalmente se realizó el montaje del sistema con ASIBOT y se intentó hacer que funcionase el sistema en el brazo. Viendo que no funcionaba de forma correcta, se prepararon dos entornos nuevos en los que tenemos el robot cartesiano, en el que sí funciona el sistema, dentro de la cocina de ASIBOT. Uno de los entornos se guía mediante cuatro esferas y otro detecta la lata de coca-cola del entorno y se guía hacia ella. Toda la programación se realizó con la ayuda del manual de usuario [32], aparte de búsquedas online.

Capítulo 4

Experimentos y resultados del sistema de control

En este capítulo vamos a mostrar los experimentos y resultados realizados con el sistema de control desarrollado en el proyecto. Se han realizado una serie de pruebas en el entorno de pruebas, el cual consiste en el robot que se mueve en el espacio. Es un espacio más propicio para comprobar el correcto funcionamiento del sistema, debido a la ausencia de entorno. Además al ser un robot cartesiano, podemos introducir directamente las coordenadas de velocidad. Y debido a que el robot se mueve por el espacio siempre puede llegar a cualquier posición del espacio con la orientación requerida.

4.1. Entorno de pruebas robot cartesiano

En este entorno hemos realizado pruebas con las que ver como afectan los parámetros del sistema al comportamiento del mismo. Esto afecta a la evolución del error en coordenadas de la imagen, posición de los motores y tiempo total para comprobar el funcionamiento. Este entorno está más controlado y es más preciso a la hora de medir, ya que aquí tenemos un cubo que "flota" por el espacio y sus posicionamientos no están tan restringidos como en el simulador de ASIBOT. Vamos a forzar al sistema a llegar a 2500 iteraciones totales para así poder comprobar el funcionamiento con

el tiempo y no parar al sistema por llegar a un error determinado. El sistema de control, en el ordenador en el que se prueba, funciona de media a 10Hz. Teniendo esto en cuenta, también se ha postulado que el sistema converge si repite el mismo valor de error durante un determinado tiempo. Este valor lo hemos establecido en 10 segundos ó 100 iteraciones si tenemos en cuenta la frecuencia. Vamos a medir en que iteración ocurre según cada caso. También vamos a medir en que iteración se pasa por un valor de error determinado. Así es como funciona por defecto el programa, y es como es más lógico hacerlo funcionar si queremos que la respuesta sea la más rápida posible. Este error es la norma euclídea del vector error como se mencionó anteriormente. Vamos a medir cuando pasa por un valor de 5 en la norma del error. En la resolución horizontal es un error del 0,0039 % suponiendo que todo el error pertenece solo a una de las features. En la resolución vertical es un error del 0,0048 % como máximo también.

4.1.1. Desde una posición sin rotación

Primero hemos realizado una serie pruebas desde una posición en la que el robot se tiene que mover para centrar las features en X y Y, y además avanzar en Z. En la posición deseada final el robot queda sin rotación. Pero el robot parte sin estar rotado, con lo que no debe de rotar en el transcurso de la simulación, o hacerlo con valores muy pequeños. Las ganancias son adimensionales, la convergencia y el paso por error es la medida de en que iteración ocurre y el error es la norma euclídea del error en píxeles. En cada tanda de simulación seguiremos un criterio ligeramente distinto hasta para encaminarnos a un funcionamiento más óptimo del sistema.

Primera simulación

La primeras pruebas las realizamos con una ganancia determinada cada vez y durante toda la simulación. Una para los motores de traslación, y un cinco por ciento de esta aplicada a los motores de

rotación. Las ganancias aplicadas junto con los resultados se muestran en la tabla 4.1.

Tabla 4.1: Parámetros y resultados del sistema

Simulación	Ganancias	Convergencia y paso por error	Error	Tiempo (s)
1	$\lambda_1 = 1$	convergencia = 1685	2,449	253,556
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 1551	4,242	
2	$\lambda_1 = 2$	convergencia = 1011	1,414	250,744
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 769	4,690	
3	$\lambda_1 = 4$	convergencia = 626	1,414	250,313
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 369	4,690	
4	$\lambda_1 = 8$	convergencia = No		252,937
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 163	4,472	

Nos produce estas gráficas comparativas para cada uno de los parámetros aplicados al sistema. En la primera gráfica se marcan los puntos de convergencia y en la segunda el paso por el valor del error.

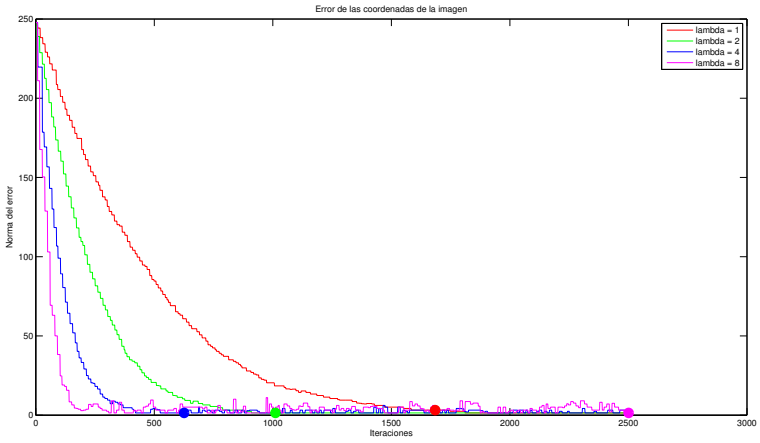

Figura 4.1: Evolución del error con el tiempo y convergencia

En esta primera gráfica (figura 4.1), podemos ver que, en general

la respuesta del error se puede asemejar a una exponencial inversa. Según aplicamos una ganancia mayor al sistema el tiempo de respuesta en general del mismo se reduce. Pero vemos que según aumentamos la ganancia, cuando la norma del error es pequeña, se producen rebotes en el error y además tenemos la ganancia de los motores de traslación asociada a los de rotación. De tal manera que según nuestro criterio al tener aplicada la ganancia máxima el sistema no converge.

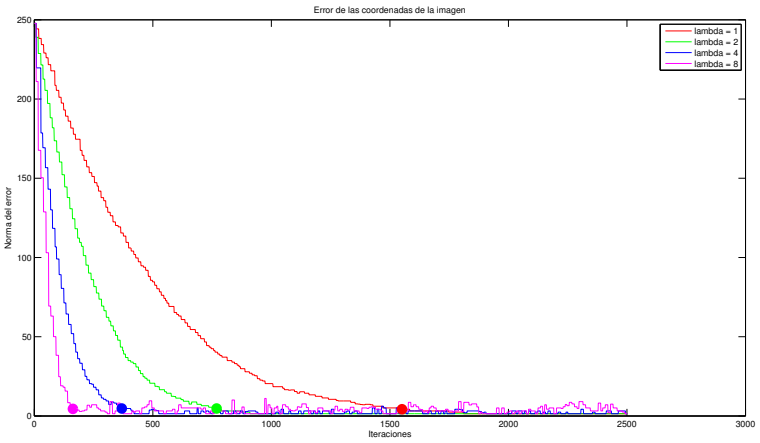

Figura 4.2: Evolución del error con el tiempo marcando paso por error

Esta segunda gráfica (figura 4.2) es como la primera pero en esta marcamos el paso por 5 del error. En este caso si que es mejor cuanto mayor sea la ganancia, de tal manera que el sistema es más rápido con $\lambda = 8$.

Segunda simulación

Para este caso hemos realizado una simulación en la que la ganancia de los motores de rotación está desligada de la de la ganancia de los motores de traslación, y está fijada en 0.05. Las ganancias y los resultados de esta simulación se muestran en la tabla 4.2. Las gráficas de error que se producen con estas ganancias se muestran a continuación. En la primera (figura 4.3) están marcados los puntos

Tabla 4.2: Parámetros y resultados del sistema

Simulación	Ganancias	Convergencia y paso por error	Error	Tiempo (s)
1	$\lambda_1 = 1$	convergencia = 1669	3,162	253,988
	$\lambda_2 = 0.05$	paso por error = 1527	4,242	
2	$\lambda_1 = 2$	convergencia = 947	1,414	253,135
	$\lambda_2 = 0.05$	paso por error = 737	4,242	
3	$\lambda_1 = 4$	convergencia = 539	1,414	252,548
	$\lambda_2 = 0.05$	paso por error = 365	4,690	
4	$\lambda_1 = 8$	convergencia = 2364	1,414	252,801
	$\lambda_2 = 0.05$	paso por error = 160	4,690	

de convergencia para cada ganancia. Podemos ver que en general se converge un poco antes que en el caso de la simulación anterior en la que teníamos las ganancias ligadas. En cuanto al error vemos que el rebote es menos pronunciado, aunque con la ganancia $\lambda = 8$ sigue existiendo, parece más cíclico. Con la ganacia $\lambda = 4$ se ha reducido bastante el rebote con respecto al caso anterior.

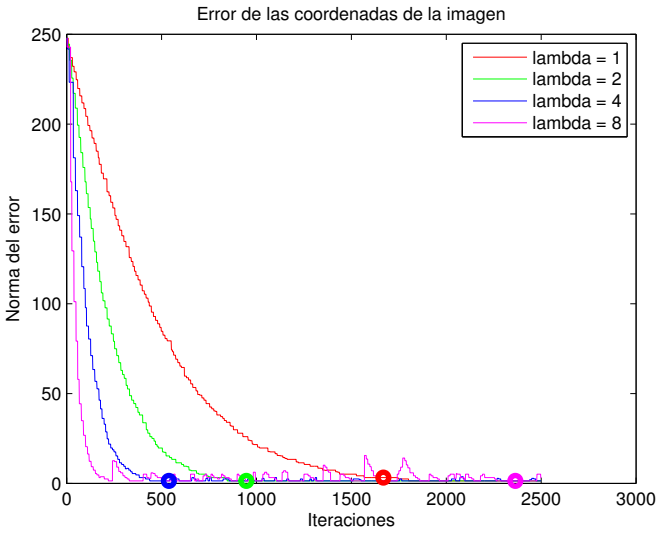

Figura 4.3: Evolución del error con el tiempo y convergencia

En cuanto al paso por el valor del error (figura 4.4), vemos que se mantiene bastante similar al caso anterior, aunque en todos los casos se disminuye ligeramente.

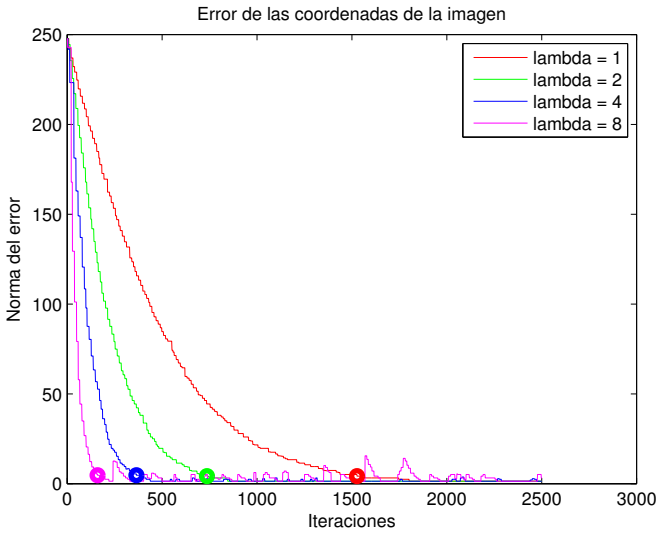

Figura 4.4: Evolución del error con el tiempo marcando paso por error

Tercera simulación

Como en el caso anterior seguía existiendo un efecto rebote cuando la ganancia es alta para una misma ganancia a las velocidades de rotación, se puede decir que la ganancia en los motores de traslación también produce este efecto. Para este caso se aplica un umbral a las ganancias, pasando estas a la mitad cuando la norma del error pase por el valor de 50. Las ganancias para la rotación está ligada con el 5 % de las ganancias de traslación. Las ganancias y los resultados de esta simulación se muestran en la tabla 4.3. En la figura 4.5 podemos observar como las convergencias en general se han hecho algo más lentas que en los otros casos de simulación, a excepción del caso de la ganancia $\lambda = 8$ en cuyo caso se ha reducido significativamente. Esto es lógico debido al umbral que hemos introducido en este grupo de simulaciones. Por otra parte se puede observar como

Tabla 4.3: Parámetros y resultados del sistema

Simulación	Ganancias	Convergencia y paso por error	Error	Tiempo (s)
1	$\lambda_1 = 1$	convergencia = 2062	5,099	253,940
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 2076	4,242	
2	$\lambda_1 = 2$	convergencia = 1525	1,414	249,395
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 1180	4,690	
3	$\lambda_1 = 4$	convergencia = 853	1,414	253,168
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 571	4,690	
4	$\lambda_1 = 8$	convergencia = 1675	1,414	251,580
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 268	4,690	

el rebote del error se ha reducido bastante en todos los casos.

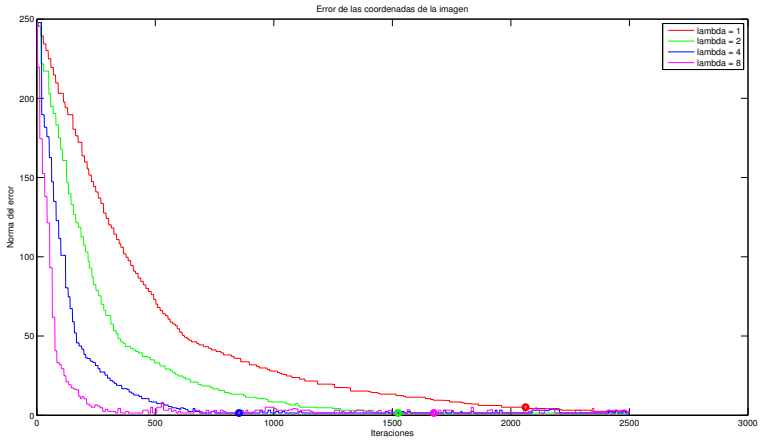

Figura 4.5: Evolución del error con el tiempo y convergencia

En cuanto al paso por el valor marcado del error (figura 4.6), observamos que en esto si que se ha hecho bastante más lento el sistema en general y con cualquier ganancia aplicada. En general se ve un corte en la tendencia en la evolución del error cuando este pasa por el valor de 50 debido al umbral aplicado.

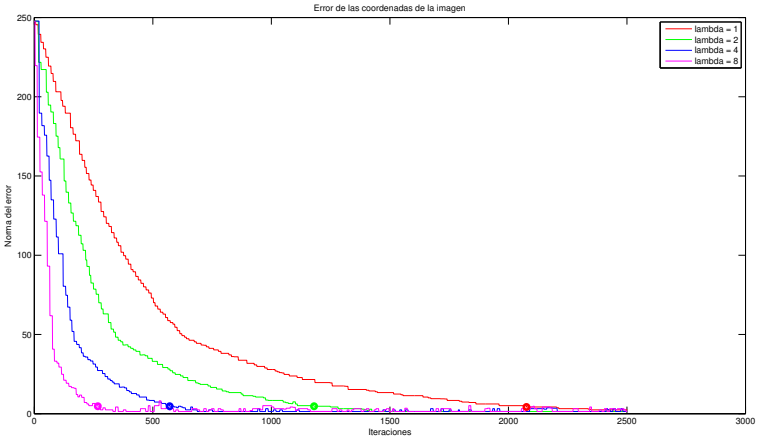

Figura 4.6: Evolución del error con el tiempo marcando paso por error

Cuarta simulación

Para esta simulación vamos a mantener el umbral de error para la reducción de ganancias aplicada en el caso anterior, pero vamos a desligar las ganancias de rotación de las de traslación, y las vamos a dejar fijas en 0,05. Las ganancias y los resultados de esta simulación se muestran en la tabla 4.4. En este caso podemos dibujar una gráfi-

Tabla 4.4: Parámetros y resultados del sistema

Simulación	Ganancias	Convergencia y paso por error	Error	Tiempo (s)
1	$\lambda_1 = 1$	convergencia = 2292	5,099	251,836
	$\lambda_2 = 0.05$	paso por error = 2297	4,690	
2	$\lambda_1 = 2$	convergencia = 1322	3,162	253,258
	$\lambda_2 = 0.05$	paso por error = 1169	4,242	
3	$\lambda_1 = 4$	convergencia = 812	1,414	250,897
	$\lambda_2 = 0.05$	paso por error = 582	4,242	
4	$\lambda_1 = 8$	convergencia = 572	1,414	251,580
	$\lambda_2 = 0.05$	paso por error = 250	4,899	

ca como en la figura 4.7. En esta gráfica podemos apreciar que el sistema en cuanto a la medición de la convergencia es bastante más

rápido que el anterior a excepción de la ganancia $\lambda = 1$ en donde es algo más lento. Sin embargo se puede apreciar como se ha disminuido en gran parte el efecto rebote hasta casi eliminarlo. En cuanto

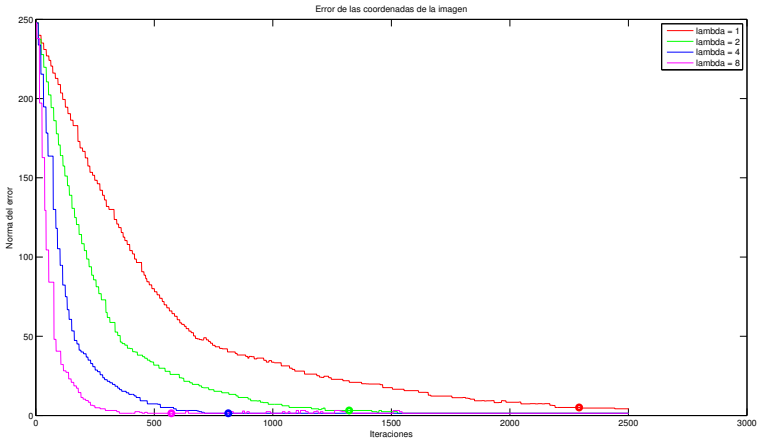

Figura 4.7: Evolución del error con el tiempo y convergencia

al paso por el error (figura 4.8), es bastante similar en cuanto a la rapidez que en el caso anterior.

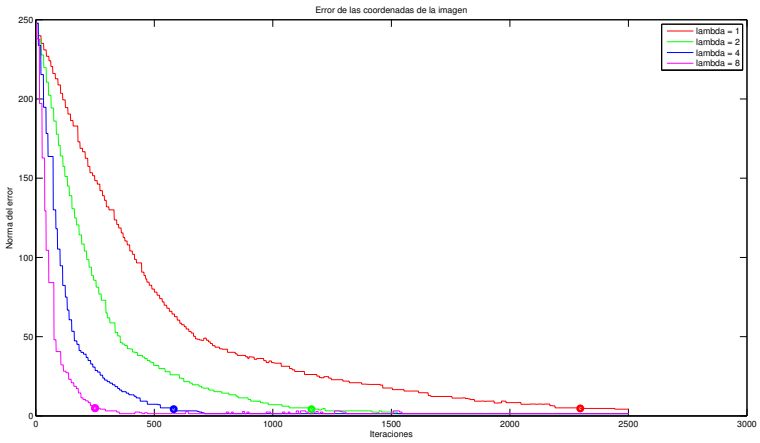

Figura 4.8: Evolución del error con el tiempo marcando paso por error

Quinta simulación

En este caso vamos a seguir aplicando un umbral a las ganancias, pero cuando la norma del error sea menor de 25. Vamos a tener ligadas las ganancias de rotación con un 5 % de las de traslación (tabla 4.5). Esta simulación nos produce la figura 4.9, donde volvemos a

Tabla 4.5: Parámetros y resultados del sistema

Simulación	Ganancias	Convergencia y paso por error	Error	Tiempo (s)
1	$\lambda_1 = 1$	convergencia = 2075	5,099	254,083
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 2162	4,690	
2	$\lambda_1 = 2$	convergencia = 1077	4,242	253,435
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 922	4,899	
3	$\lambda_1 = 4$	convergencia = 832	1,414	251,340
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 500	4,690	
4	$\lambda_1 = 8$	convergencia = 509	1,414	252,420
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 231	3,162	

tener un poco de rebote al llegar a errores bajos, pero menor que en los casos anteriores en los que teníamos las ganancias de rotación ligadas a las de traslación. La convergencia se reduce también con

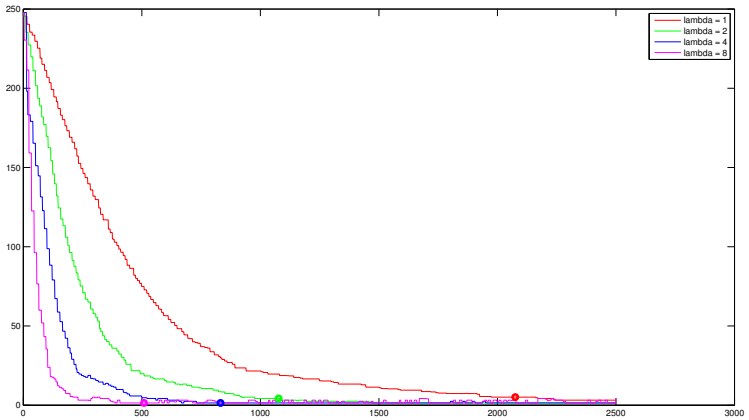

Figura 4.9: Evolución del error con el tiempo y convergencia

respecto a los casos anteriores de ganancias ligadas. El paso por error lo vemos en la figura 4.10.

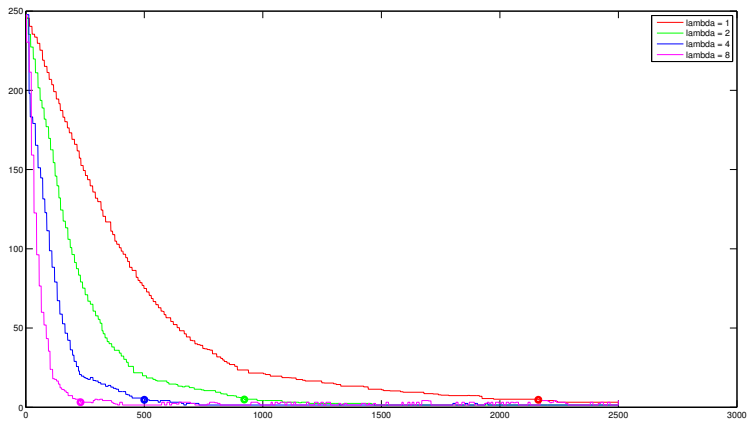

Figura 4.10: Evolución del error con el tiempo marcando paso por error

Sexta simulación

Para este caso vamos a seguir aplicando el umbral en 25, sin embargo las ganancias de rotación las vamos dejar fijas en 0,05. Las ganancias y resultados los vemos en la tabla 4.6. Con estos resultados

Tabla 4.6: Parámetros y resultados del sistema

Simulación	Ganancias	Convergencia y paso por error	Error	Tiempo (s)
1	$\lambda_1 = 1$	convergencia = 1807	7,348	253,749
	$\lambda_2 = 0.05$	paso por error = 2080	4,242	
2	$\lambda_1 = 2$	convergencia = 1191	2,449	249,054
	$\lambda_2 = 0.05$	paso por error = 937	4,899	
3	$\lambda_1 = 4$	convergencia = 614	3,162	253,256
	$\lambda_2 = 0.05$	paso por error = 502	4,690	
4	$\lambda_1 = 8$	convergencia = 432	1,414	251,018
	$\lambda_2 = 0.05$	paso por error = 199	4,690	

vemos la figura 4.11, donde podemos ver que ha vuelto a reducirse

casi por completo el rebote del error y además al aplicar el umbral más tarde el sistema es más rápido. En el caso de la ganancia máxima $\lambda = 8$ hemos conseguido los mejores resultados en esta simulación, en general en este caso vemos que funciona mejor para ganancias altas que para bajas. Para ver el paso por el error, vemos la figura

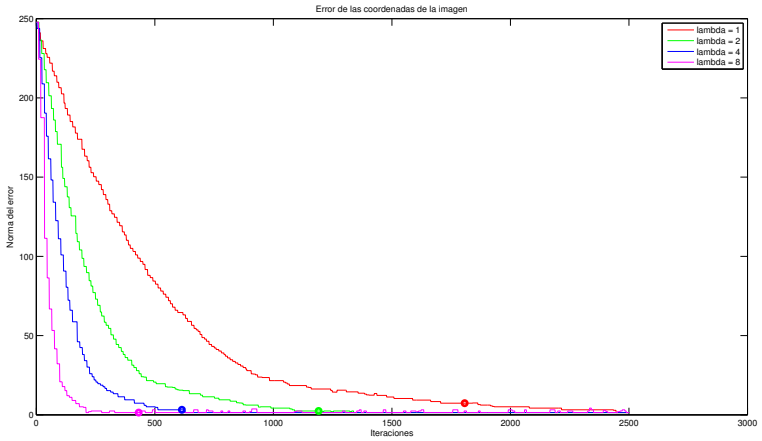

Figura 4.11: Evolución del error con el tiempo y convergencia

4.12, en donde vemos que es un poco más lento que si no se aplica umbral pero es más rápido que el umbral en 50. Después de todas

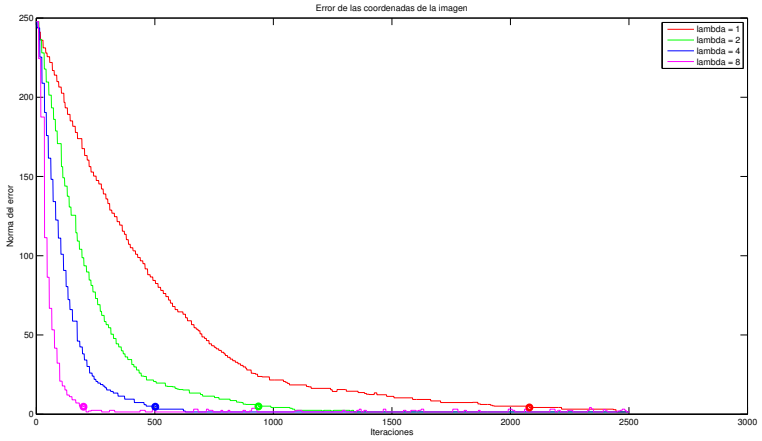

Figura 4.12: Evolución del error con el tiempo marcando paso por error

estas simulaciones, podemos decir que con el criterio de convergencia vemos la estabilidad con la que evoluciona el error, y con el paso por el error vemos la rapidez a la hora de disminuir el error. El sistema es más rápido cuanta mayor ganancia apliquemos, pero llegados a un punto esta no puede ser muy grande si se pretende tener resultados precisos o ajustados. Parece que el resultado que conjuga la mayor precisión junto con velocidad se produce con la mayor ganancia y en el último experimento.

4.1.2. Desde una posición rotada

Ahora vamos a repetir alguna de las simulaciones partiendo desde la misma posición que en las simulaciones anteriores en la que además ahora el robot se encuentre rotado. Por lo tanto el robot tendrá que rotar a la par que avanzar en los distintos ejes para llegar a la posición final en la que no tiene rotación y se encuentra centrado respecto a las features. En este caso también mostraremos alguna gráfica de la evolución de los motores tanto de traslación como de rotación para comprobar el correcto funcionamiento del sistema. Vamos a realizar directamente las simulaciones con y sin umbral, pero fijando este en 25 y sin ver el umbral en 50, ya que este produce que el sistema sea más lento y tampoco produce mucha ganancia en estabilidad. Para poder partir de la misma posición rotada en cada experimento se ha creado un pequeño programa que envía unas velocidades y mediante la función pause de MATLAB, se mantienen durante un determinado tiempo y luego se para al robot. Una vez alcanzada esa posición se inicia el sistema de Visual Servoing. Partiremos con el robot teniendo una rotación de 10^{o} en cada eje.

Primera simulación

Siguiendo los pasos anteriores primero vamos a realizar esta simulación con las ganancias de rotación ligadas a las de traslación por un 5 %, y sin aplicar un umbral de reducción de ganancias. Parámetros en la tabla 4.7. Tenemos la figura 4.13 en la que podemos

Tabla 4.7: Parámetros y resultados del sistema

Simulación	Ganancias	Convergencia y paso por error	Error	Tiempo (s)
1	$\lambda_1 = 1$	convergencia = 1641	4,690	250,748
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 1540	4,690	
2	$\lambda_1 = 2$	convergencia = 1091	1,414	250,616
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 743	4,690	
3	$\lambda_1 = 4$	convergencia = 636	1,414	249,250
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 361	4,242	
4	$\lambda_1 = 8$	convergencia = No		252,684
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 188	1,414	

ver la convergencia. Partimos de un error mucho más elevado ya que está rotado el robot. Se puede ver que el error mientras decrece lo

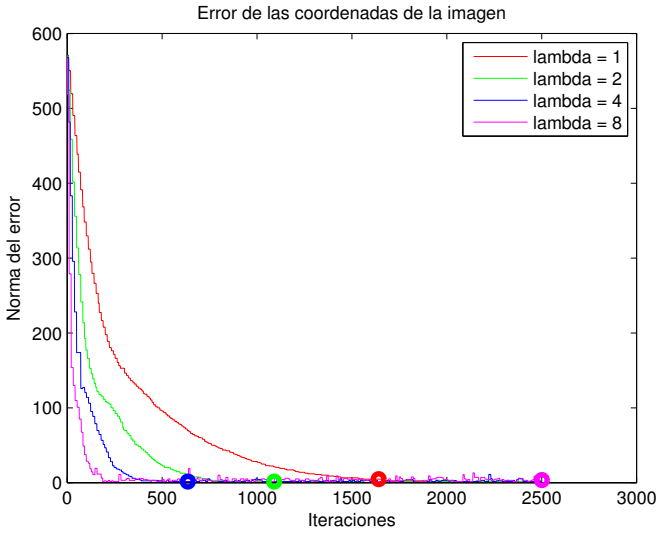

Figura 4.13: Evolución del error con el tiempo y convergencia

hace de una manera un tanto inestable, porque ahora está rotando a la par que avanza y tener las ganancias de rotación ligadas a las de traslación hace que sea un poco más pronunciado este efecto. En cuanto al paso por el valor de la norma del error en el valor de 5, podemos ver la figura 4.14. Aquí observamos que a mayor

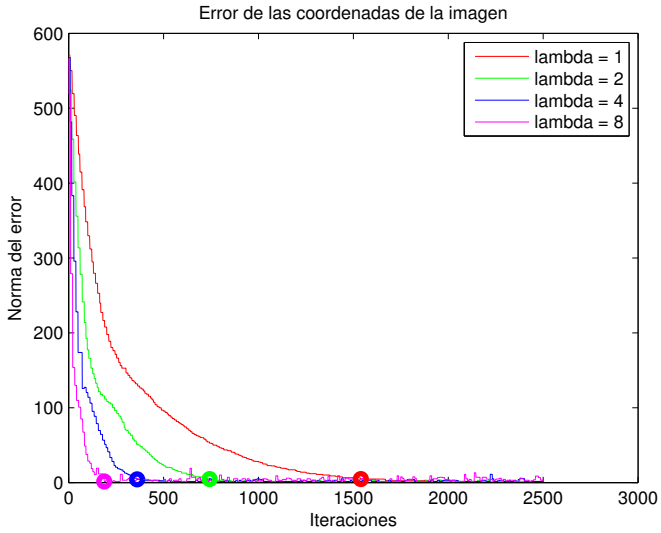

Figura 4.14: Evolución del error con el tiempo marcando paso por error

ganancia antes pasa por el valor de error marcado. A alta ganancia se producen rebotes en el error. El error inicialmente disminuye con una gran rapidez ya que el robot se encuentra rotado.

Segunda simulación

Ahora no vamos aplicar umbral a las ganancias, dejando fija la velocidad de rotación (tabla 4.8). En la siguiente gráfica (figura

Tabla 4.8: Parámetros y resultados del sistema

Simulación	Ganancias	Convergencia y paso por error	Error	Tiempo (s)
1	$\lambda_1 = 1$	convergencia = 1948	1,414	249,813
	$\lambda_2 = 0.05$	paso por error = 1636	4,690	
2	$\lambda_1 = 2$	convergencia = 1088	1,414	245,576
	$\lambda_2 = 0.05$	paso por error = 803	4,690	
3	$\lambda_1 = 4$	convergencia = 1214	1,414	250,599
	$\lambda_2 = 0.05$	paso por error = 967	3,741	
4	$\lambda_1 = 8$	convergencia = 2236	1,414	251,145
	$\lambda_2 = 0.05$	paso por error = 1047	3,162	

4.15) vemos la convergencia. La evolución del error tiene un comportamiento muy errático. La rapidez la vemos en la figura 4.16.

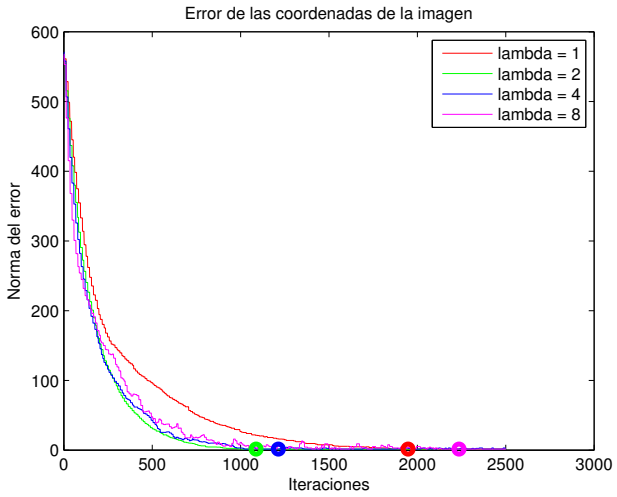

Figura 4.15: Evolución del error con el tiempo y convergencia

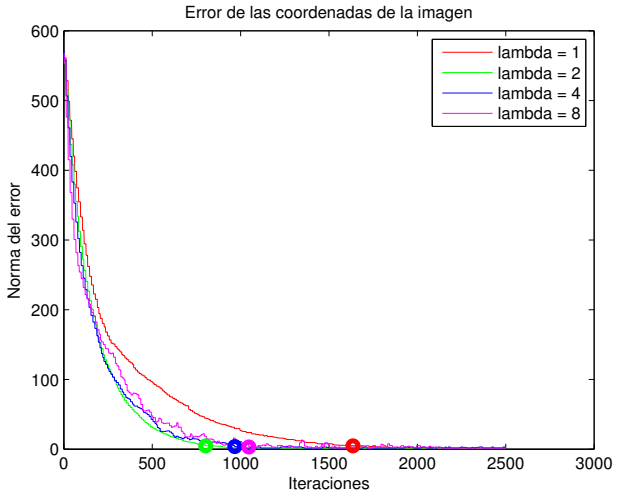

Figura 4.16: Evolución del error con el tiempo marcando paso por error

Se observa que tener la ganancia de rotación desligada de la de traslación, produce que a altas ganancias el sistema tenga una evolución del error muy irregular. Esto ocurre por que el sistema no tiene suficiente velocidad de rotación y se traduce en un sistema más lento

y menos estable.

Tercera simulación

Ahora en esta simulación vamos a aplicar un umbral a las ganancias para reducirlas a la mitad una vez que la norma del error pase por el valor de 25. Además vamos a tener ligadas las ganancias de rotación a las de traslación por un 5 % del valor de estas. Vamos a observar en conjunto el tener un umbral con tener ganancias altas en rotación al tener estas ligadas a las de traslación cuando se parte de una posición rotada. Las ganancias aplicadas y los resultados obtenidos se introducen en la tabla 4.9.

Tabla 4.9: Parámetros y resultados del sistema

Simulación	Ganancias	Convergencia y paso por error	Error	Tiempo (s)
1	$\lambda_1 = 1$	convergencia = 1870	8,602	249,604
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 2253	4,242	
2	$\lambda_1 = 2$	convergencia = 1120	4,690	250,747
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 1019	4,690	
3	$\lambda_1 = 4$	convergencia = 622	3,612	250,103
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 521	3,612	
4	$\lambda_1 = 8$	convergencia = 608	1,414	253,187
	$\lambda_2 = 0.05 * \lambda_1$	paso por error = 258	4,242	

La primera de las gráficas que nos produce es la que se puede ver en la figura 4.17 y es donde tenemos marcados los puntos de convergencia. En este caso vemos que el error evoluciona más acorde con la ganancia aplicada al sistema, siendo más rápido cuanto mayor ganancia apliquemos. La convergencia se produce antes que en el caso anterior en general, pero un poco después que en el primer caso.

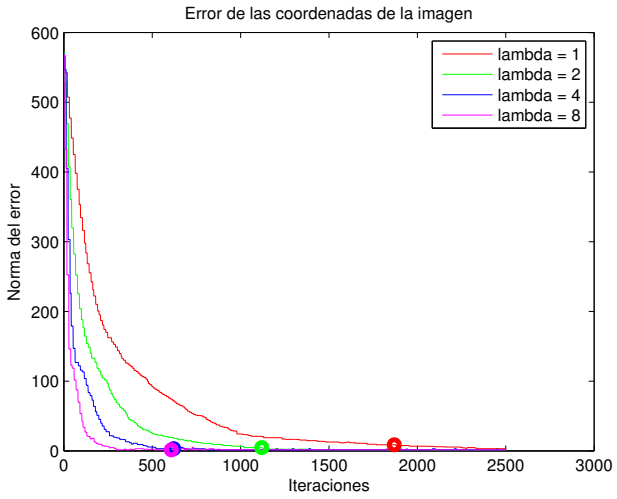

Figura 4.17: Evolución del error con el tiempo y convergencia

La gráfica en la que vemos el paso por el error es la figura 4.18, donde vemos que el paso por el error es acorde a la ganancia aplicada.

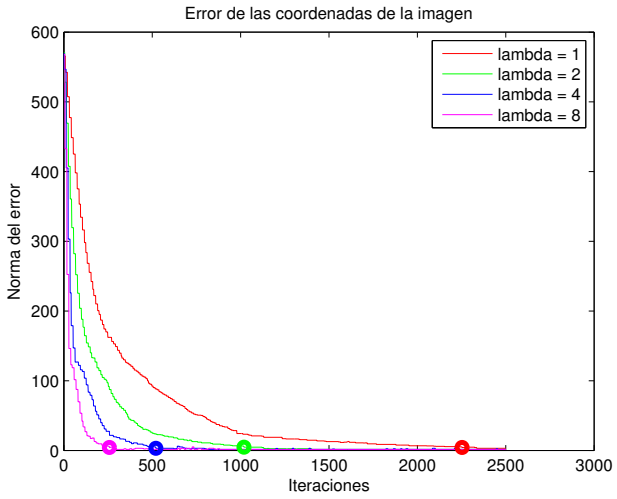

Figura 4.18: Evolución del error con el tiempo marcando paso por error

Cuarta simulación

En esta simulación vamos aplicar un umbral a las ganancia cuando la norma del error pase por el valor de 25, pero vamos a desligar

la ganancia de las rotaciones que vamos a fijar en 0,05. Esto debería disminuir el efecto de rebote del error del sistema. Los parámetros se incluyen en la tabla 4.10. En la figura 4.19, lo primero que podemos

Tabla 4.10: Parámetros y resultados del sistema

Simulación	Ganancias	Convergencia y paso por error	Error	Tiempo (s)
1	$\lambda_1 = 1$	convergencia = 1615	10,295	246,307
	$\lambda_2 = 0.05$	paso por error = 1918	4,242	
2	$\lambda_1 = 2$	convergencia = 1402	3,741	250,050
	$\lambda_2 = 0.05$	paso por error = 1079	4,899	
3	$\lambda_1 = 4$	convergencia = 1172	1,414	253,690
	$\lambda_2 = 0.05$	paso por error = 935	4,690	
4	$\lambda_1 = 8$	convergencia = 1509	1,414	253,811
	$\lambda_2 = 0.05$	paso por error = 918	4,242	

observar es que se repite el comportamiento de la segunda simulación. Lo que produce que a altas ganancias el sistema funcione peor que a bajas ganancias. A altas ganancias el sistema muestra una evolución un poco errática del error. En la figura 4.20 pode-

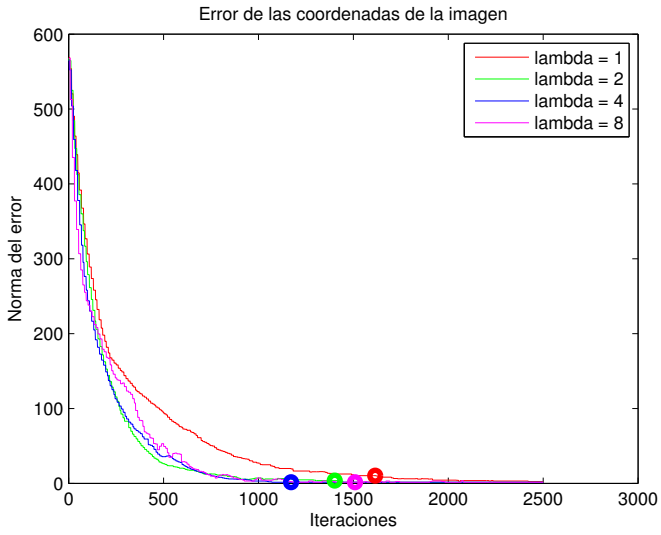

Figura 4.19: Evolución del error con el tiempo y convergencia

mos ver que esto también afecta a la rapidez con la que el sistema pasa por el error marcado, haciéndose más lento a mayor ganancia. Después de ver estas cuatro simulaciones y ver los resultados mostra-

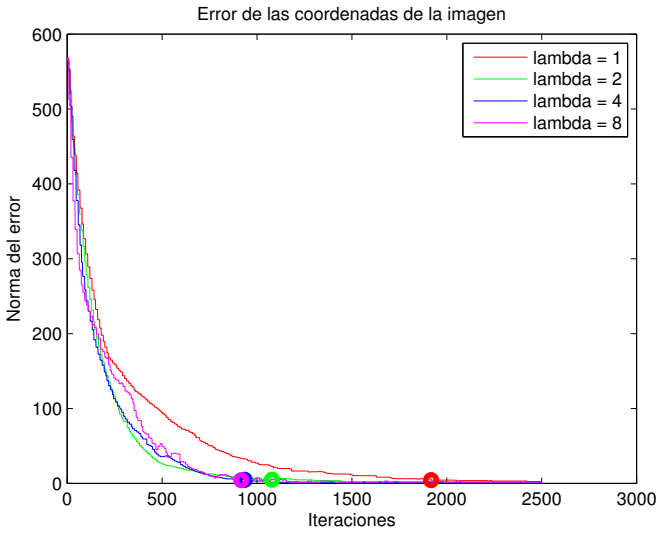

Figura 4.20: Evolución del error con el tiempo marcando paso por error

dos, podemos decir que en el caso de que el robot parta desde una posición con rotación y esto le obligue a avanzar y rotar, tener aplicada una baja ganancia de rotación y fija, produce que el sistema sea más lento debido a que rota demasiado lento mientras avanza. En las simulaciones con las ganancias de rotación ligadas a las de traslación se ve un comportamiento mucho mejor en el error del sistema. En estos casos le damos la suficiente ganancia a la rotación para que el robot rote y avance, consiguiendo antes converger y pasar por un error marcado. El resultado más óptimo parece el producido en la tercera simulación, es decir, aplicando un umbral a las ganancias pero teniéndolas ligadas. Para ese caso y con la ganacia de $\lambda = 4$ el sistema sufre la evolución en los motores que se puede ver la figura 4.21. En dichas gráficas podemos comprobar como el sistema al final llega a una determinada posición que le hemos marcado en (X, Y, Z) y además en esa posición los motores de rotación están a cero, es decir, el robot en su posición final se encuentra sin rotación en

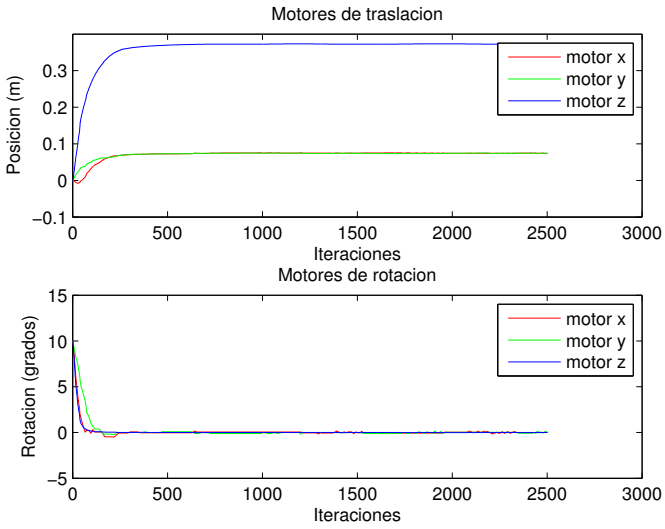

Figura 4.21: Evolución de la posición de los motores con el tiempo

ningún eje.

4.2. Estudio de la profundidad

En este apartado vamos a ver como influye la profundidad entre la cámara y los puntos al sistema de control y su evolución con el tiempo. Procesar la Jacobiana de la imagen requiere el conocimiento de Z_i, es decir, de la profundidad de cada punto. En la simulaciones anteriores hemos supuesto una profundidad conocida y fijada en uno. En la realidad esto no es posible, pero como vamos a ver a continuación el sistema es bastante tolerable a errores en Z. Vamos a realizar una simulación desde el mismo punto de partida de las anteriores pero rotando el robot 10^{o} por eje. Para estás simulaciones, vamos a tener una ganancia de traslación de 4 y la de rotación va a estar ligada al 5 % de la de traslación. Ambas ganancias con un umbral de error de 25 tras el cual se reducen a la mitad. Las profundidades aplicadas son iguales para cada punto, aunque se pueden especificar por separado. La tabla 4.11 contiene los resultados de la simulación igual que en los casos anteriores. En la figura 4.22, pode-

Tabla 4.11: Profundidad aplicada y resultados del sistema

Simulación	Profundidad	Convergencia y paso por error	Error	Tiempo (s)
1	$Z_i = 1$	convergencia = 651	1,414	252,437
		paso por error = 536	4,000	
2	$Z_i = 2$	convergencia = 403	1,414	252,174
		paso por error = 243	4,690	
3	$Z_i = 4$	convergencia = 1516	1,414	250,898
		paso por error = 290	3,612	
4	$Z_i = 8$	convergencia = 677	1,414	247,408
		paso por error = 238	3,741	

mos ver que la profundidad que hace que el sistema converja antes, es la profundidad Z = 2. Se observa un comportamiento similar que a la hora de aplicar diferentes ganancias y según aumentamos, menos suave es la gráfica. También se ve que la norma del error evoluciona

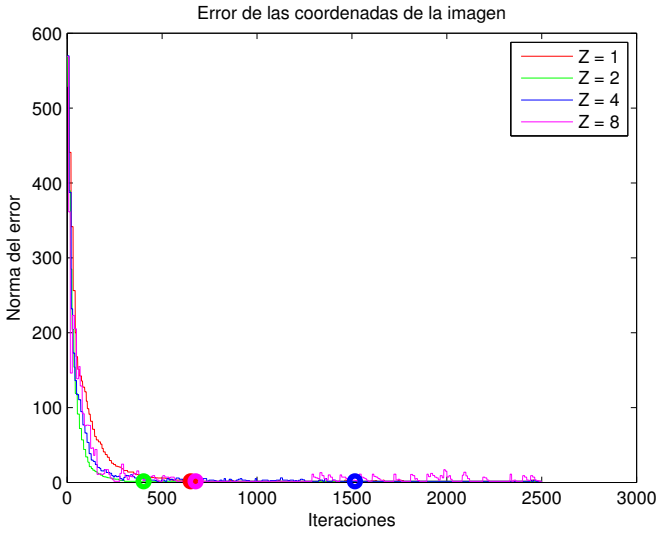

Figura 4.22: Convergencia según profundidad

de forma más suave con Z=1 y Z = 2, mientras que con profundidades mayores la trayectoria del error es irregular. Al aplicar grandes profundidades y el sistema encontrarse ya en errores bajos se pro-

ducen los rebotes que se observaban en los estudios anteriores de la ganancia. Pero estos rebotes son algo menores y además no son tan caóticos como los producidos en los estudios anteriores. En la figura 4.23, podemos ver que con la profundidad con la que el sistema pasa más rápido por el error marcado es prácticamente la misma para Z=2 y Z=8. Además no hay una gran diferencia entre cualquiera de las profundidades aplicadas al contrario de como ocurría antes cuando variamos las ganancias. Para ver mejor aún la diferencia en-

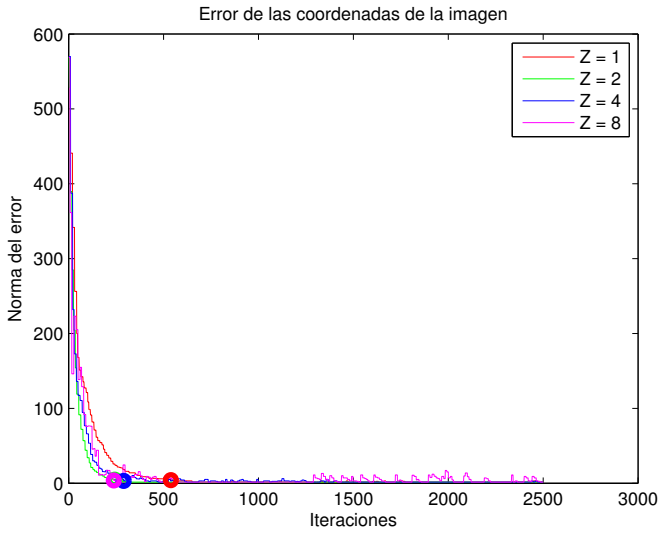

Figura 4.23: Rapidez según profundidad

tre las distintas profundidades vamos a mostrar las gráficas de las evoluciones de los motores para Z = 2, que es la que parece que obtiene un mejor resultado y para Z = 8 es la que muestra el peor comportamiento o por lo menos más irregular. La evolución de los motores para Z = 2 la podemos ver en la figura 4.24. En esta gráfica se puede observar un comportamiento de los motores bastante suave con el tiempo. En la figura 4.25 donde Z = 8, se puede ver que si bien los motores de rotación tienen un funcionamiento aceptable, los motores de traslación muestran un gran pico en la gráfica y tienen movimientos bruscos. Muestran una trayectoria irregular con el tiempo, hasta que se estabilizan en sus correspondientes posi-

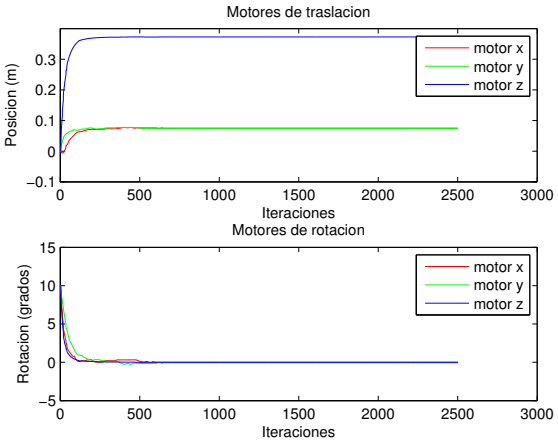

Figura 4.24: Evolución de los motores con profundidad Z = 2

ciones. La profundidad solo afecta al movimiento de traslación del

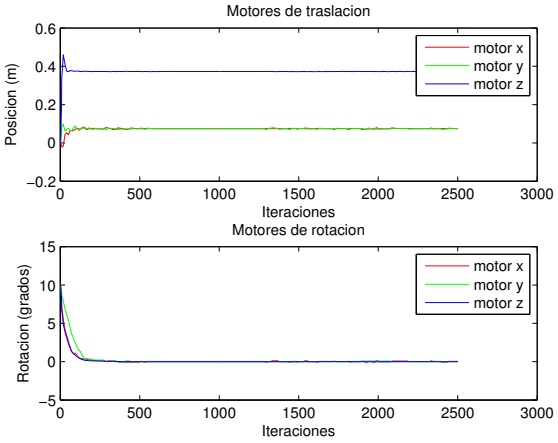

Figura 4.25: Evolución de los motores con profundidad Z = 8

robot, ya que la parte de la Jacobiana en la que interviene solo multiplica a las velocidades de traslación y no a las de rotación. Viendo estos resultados podemos decir que el sistema es bastante tolerante a errores en la profundidad. El sistema funcionará con profundidades erróneas, producirá un movimiento lento y más lineal si se subestima y un movimiento más rápido e irregular si se sobrestima. Una forma de funcionamiento bastante aceptada, consiste en aplicar una

profundidad igual a la que se desea que exista al finalizar la acción de control entre la cámara y los features.

4.3. Experimentos de posición

Ahora vamos a ver el correcto funcionamiento del sistema con el experimento más intuitivo. Para ello vamos a hacer que el robot parta de diferentes posiciones en el espacio con o sin rotación y veremos si nos conduce al mismo punto del espacio con respecto a las features. Vamos a definir la matriz de coordenadas deseadas de los puntos features que queremos tener al final, dicha matriz es la siguiente:

$$pStar = \begin{pmatrix} 440 & 440 & 840 & 840 \\ 312 & 712 & 712 & 312 \end{pmatrix}$$

Vamos a realizar la primera simulación partiendo desde el origen del sistema, en donde el robot no se encuentra centrado respecto de las features. El vector de posición del robot es de la forma $P = (p_x(m), p_y(m), p_z(m), w_x(^\circ), w_y(^\circ), w_z(^\circ))$.

1. Partimos desde la siguiente posición en esta simulación.

   $$P = (0, 0, 0, 0, 0, 0)$$

   Al finalizar la simulación llegamos a la posición.

   $$pfin = (0.074, 0.074, 0.613, 0.027, -0.028, -0.01)$$

   En está posición final el robot ve las siguientes coordenadas de la cámara.

   $$pFin = \begin{pmatrix} 442 & 442 & 839 & 839 \\ 315 & 711 & 711 & 314 \end{pmatrix}$$

2. En la segunda simulación vamos a partir de la posición.

$$P = (-0.1, -0.1, 0, 10.013, 10.013, 15.020)$$

Una vez finalizada la acción de control el robot se encuentra en la posición.

$$pfin = (0.074, 0.073, 0.613, 0.003, -0.006, 0.002)$$

En donde la cámara del robot obtiene las siguientes coordenadas de los features.

$$pFin = \begin{pmatrix} 440 & 440 & 839 & 839 \\ 314 & 712 & 712 & 314 \end{pmatrix}$$

3. En la tercera simulación el robot parte de la posición.

$$P = (-0.038, 0.34, 0.235, 0.007, -0.033, -24.144)$$

Una vez finalizada la acción de control el robot se encuentra en la posición.

$$pfin = (0.074, 0.075, 0.614, 0.030, 0.005, -0.012)$$

En donde la cámara del robot obtiene las siguientes coordenadas de los features.

$$pFin = \begin{pmatrix} 441 & 441 & 840 & 841 \\ 311 & 710 & 710 & 311 \end{pmatrix}$$

4. En la cuarta simulación el robot parte de la posición.

$$P = (0.152, 0.153, 0.653, 6.384, 6.357, -0.01)$$

Una vez finalizada la acción de control el robot se encuentra en la posición.

$$pfin = (0.075, 0.074, 0.614, 0.016, -0.017, -0.003)$$

En donde la cámara del robot obtiene las siguientes coordenadas de los features.

$$pFin = \begin{pmatrix} 438 & 438 & 838 & 838 \\ 311 & 711 & 711 & 311 \end{pmatrix}$$

5. En la quinta simulación el robot parte de la posición.

$$P = (-0.488, 0.552, 0, 17.500, 13.563, 16.615)$$

Una vez finalizada la acción de control el robot se encuentra en la posición.

$$pfin = (0.073, 0.074, 0.614, -0.002, -0.015, 0.000)$$

En donde la cámara del robot obtiene las siguientes coordenadas de los features.

$$pFin = \begin{pmatrix} 442 & 442 & 841 & 841 \\ 311 & 710 & 710 & 311 \end{pmatrix}$$

Una vez realizadas estas simulaciones vemos que el sistema nos lleva consistentemente a la misma posición final teniendo en cuenta la tolerancia aplicada mediante el error, desde cualquier punto del que parta el robot y con diferentes orientaciones. Con esto se muestra el hecho de que unas determinadas coordenadas de los features en la imagen solo se pueden obtener desde una posición determinada si los

features se encuentran fijos en el entorno. Si los features se mueven por el entorno el robot siempre quedará en la misma posición relativa con respecto a estos.

Capítulo 5

Conclusiones

Una vez finalizado el proyecto, se presentan a continuación las conclusiones, teniendo en cuenta los objetivos y las limitaciones en cuanto al conocimiento previo sobre la materia. Se ha conseguido una muy buena plataforma de pruebas para el sistema de control, mediante un escueto entorno y con un sencillo robot cartesiano. En dicha plataforma se ha conseguido un funcionamiento satisfactorio del sistema de control Visual Servoing, además de poder probar todos los parámetros y así entender como afectan al funcionamiento del mismo.

Finalmente se ha introducido el robot cartesiano en el entorno de la cocina de ASIBOT, consiguiendo que el robot sea capaz de guiarse hacia la lata de coca cola que se encuentra en la encimera o que se guíe mediante esferas. El sistema de control funciona de manera satisfactoria, llevando al robot a la misma posición relativa con respecto a los features independientemente de la posición de partida y de su orientación. Se ha conseguido introducir un sistema de guiado por visión dentro del repositorio, de tal manera que el sistema se puede aplicar al robot que se quiera simular, siempre que se realice la adaptación necesaria en su control. Podemos decir que los objetivos del proyecto se han cumplido de manera satisfactoria.

5.1. Desarrollos futuros

ASIBOT tiene ciertas limitaciones al tener un grado de libertad menos que lo óptimo, además de su configuración propicia a singularidades. Lo primero que hay que destacar en cuanto a posibles desarrollos futuros está la cuestión de añadir un grado de libertad más a ASIBOT de tal manera que tenga 6. Esto permitiría al robot tener una configuración mediante la cual podría llegar a los puntos del espacio dentro de su alcance con mejores orientaciones. Esta mejora sería esencial a la hora de implementar en ASIBOT sistemas Visual Servoing.

Otra mejora posible sería que los puntos usados como features, en vez de ser todos del mismo color, cada uno sea de un color diferente. Así se consigue eliminar el algoritmo de ordenación de los puntos, ya que según el color cada punto tendría un orden determinado. Esta mejora permitiría ganar algo de velocidad de procesamiento y sobretodo que el robot, en su efector final, podría tener cualquier rotación al no tener los impedimentos del algoritmo de ordenación.

Una buena opción sería desarrollar o incluir un estimador de profundidad para las features, pudiendo ser este conseguido mediante la aplicación de técnicas de visión estándar. Si se conoce la matriz de parámetros intrínsecos de la cámara se pueden aplicar técnicas de visión estéreo entre imágenes consecutivas. Otra opción seria mediante la aplicación de las medidas del robot y del movimiento de la imagen. Mediante el estimador se conseguiría un funcionamiento ajustado a la profundidad de las features con respecto a la cámara a cada paso.

Bibliografía

[1] J. Ingenieros, *El hombre mediocre/Mediocre man*. Linkgua digital, 2010.

[2] R. A. Española., "Diccionario de la lengua española." `http://lema.rae.es/drae/?val=tecnologia`, 2013. [Internet; descargado 26-junio-2013].

[3] R. A. Española., "Diccionario de la lengua española." `http://lema.rae.es/drae/?val=rob%C3%B3tica`, 2013. [Internet; descargado 26-junio-2013].

[4] A. J. Lara and A. H. García, "Estadísticas y otros registros sobre discapacidad en españa," *Política y sociedad*, vol. 47, no. 1, pp. 165–174, 2010.

[5] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, vol. 73. Springer, 2011.

[6] B. Siciliano and L. Sciavicco, *Robotics: modelling, planning and control*. Springer Verlag, 2009.

[7] T. Kroger, B. Finkemeyer, and F. M. Wahl, "A task frame formalism for practical implementations," in *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, vol. 5, pp. 5218–5223, IEEE, 2004.

[8] N. Hogan, "Impedance control: An approach to manipulation: Part illapplications," *Journal of dynamic systems, measurement, and control*, vol. 107, no. 2, p. 17, 1985.

[9] Wikipedia, "Visual servoing — wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Visual_Servoing&oldid=543379544`, 2013. [Online; accessed 9-July-2013].

[10] A. J. Huete, J. G. Victores, S. Martinez, A. Giménez, and C. Balaguer, "Personal autonomy rehabilitation in home environments by a portable assistive robot," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 42, no. 4, pp. 561–570, 2012.

[11] C. Balaguer, A. Jardón, C. Monje, F. Bonsignorio, M. Stoelen, S. Martinez, and J. Victores, "Sultan: Simultaneous user learning and task execution, and its application in assistive robotics," in *Workshop on New and Emerging Technologies in Assistive Robotics at IROS 2011* (K. WadaEditors, ed.), pp. 6–8, 2011.

[12] R. Diankov, *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010.

[13] Wikipedia, "Socket de internet — wikipedia, la enciclopedia libre." `http://es.wikipedia.org/w/index.php?title=Socket_de_Internet&oldid=66520404`, 2013. [Internet; descargado 26-julio-2013].

[14] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: yet another robot platform," *International Journal on Advanced Robotics Systems*, vol. 3, no. 1, pp. 43–48, 2006.

[15] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Incorporated, 2008.

[16] Y. Shirai and H. Inoue, "Guiding a robot by visual feedback in assembling tasks," *Pattern Recognition*, vol. 5, no. 2, pp. 99–108, 1973.

[17] G. J. Agin, *Real time control of a robot with a mobile camera.* SRI International, 1979.

[18] S. Hutchinson, G. D. Hager, and P. I. Corke, "A tutorial on visual servo control," *Robotics and Automation, IEEE Transactions on*, vol. 12, no. 5, pp. 651–670, 1996.

[19] F. Chaumette, S. Hutchinson, *et al.*, "Visual servo control, part ii: Advanced approaches," *IEEE Robotics and Automation Magazine*, vol. 14, no. 1, pp. 109–118, 2007.

[20] P. I. Corke, "Visual control of robot manipulators-a review," *Visual servoing*, vol. 7, pp. 1–31, 1993.

[21] A. Cesetti, E. Frontoni, A. Mancini, P. Zingaretti, and S. Longhi, "A vision-based guidance system for uav navigation and safe landing using natural landmarks," in *Selected papers from the 2nd International Symposium on UAVs, Reno, Nevada, USA June 8–10, 2009*, pp. 233–257, Springer, 2010.

[22] Y. Aloimonos, *Active perception.* Psychology Press, 2013.

[23] A. Sanderson and L. Weiss, "Image-based visual servo control using relational graph error signals," *Proc. ieee*, vol. 1074, 1980.

[24] R. Jarvis, "A perspective on range finding techniques for computer vision," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 2, pp. 122–139, 1983.

[25] A. C. Sanderson and L. E. Weiss, "Adaptive visual servo control of robots," *Robot Vision*, vol. 107, p. 116, 1983.

[26] S. B. Skaar, W. H. Brockman, and R. Hanson, "Camera-space manipulation," *The International journal of robotics research*, vol. 6, no. 4, pp. 20–32, 1987.

[27] K. Hashimoto, T. Kimoto, T. Ebine, and H. Kimura, "Manipulator control with image-based visual servo," in *Robotics and*

Automation, 1991. Proceedings., 1991 IEEE International Conference on, pp. 2267–2271, IEEE, 1991.

[28] F. Chaumette, P. Rives, and B. Espiau, "Positioning of a robot with respect to an object, tracking it and estimating its velocity by visual servoing," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pp. 2248–2253, IEEE, 1991.

[29] S. S. BUENO, J. R. Azinheira, J. RAMOS Jr, E. C. d. PAIVA, P. Rives, A. Elfes, J. R. Carvalho, G. F. Silveira, *et al.*, "Project aurora: Towards an autonomous robotic airship," 2010.

[30] A. Krupa, J. Gangloff, C. Doignon, M. F. de Mathelin, G. Morel, J. Leroy, L. Soler, and J. Marescaux, "Autonomous 3-d positioning of surgical instruments in robotized laparoscopic surgery using visual servoing," *Robotics and Automation, IEEE Transactions on*, vol. 19, no. 5, pp. 842–853, 2003.

[31] J.-Y. Bouguet, "Camera calibration toolbox for matlab," 2004.

[32] M. U. Guide, "The mathworks," *Inc., Natick, MA*, vol. 5, 2013.

Anexo: Presupuesto de realización del proyecto

A continuación se presenta el presupuesto de realización del proyecto como se estipula en la universidad Carlos III de Madrid.

1. Autor: Álvaro Martínez Estradé.
2. Departamento: Ingeniería de Sistemas y Automática.
3. Descripción del proyecto:Desarrollo de módulo de Visual Servoing para el repositorio Open Source ASIBOT.
   - Duración: 9 meses
   - Tasa de costes indirectos: 20 %
4. Desglose del presupuesto:

Coste de los equipos

Descripción	Coste (€)	% de uso	Duración (meses)	Periodo de depreciación(meses)	Coste atribuible (€)
Ordenador portátil	700	80	9	60	84

Tabla 1: Coste de los equipos

Coste de los equipos total 84€

Amortización de los equipos = $\frac{A}{B}.C.D$

A = Número de meses de uso del equipo.

B = Periodo de depreciación.

C = Coste del equipo.

D = Porcentaje de uso del equipo.

Costes laborales

Tabla 2: Costes laborales

Apellidos y nombre	Categoría	Dedicación (horas)	Coste por hora (€)	Coste (€)
González Víctores Juan	Ingeniero senior	30	40	1200
Jardón Huete Alberto	Ingeniero senior	6	40	240
Martínez Estradé Álvaro	Ingeniero	250	20	5000

Costes laborales totales 6440€

Costes indirectos = 0.2 (Costes laborales + Amortizaciones)

Costes finales

Tabla 3: Costes laborales

Tipo de coste	Coste total (€)
Amortizaciones	84
Costes laborales	6440
Costes indirectos	1304,8
Suma total	**7828,8**